COURSE PACK FOR

Subject Title: - DBMS-II (CBCS 2018 course)

COURSE CODE	: 302
COURSE	: BCA
SEMESTER	: III
YEAR	: 2020-21

Course Instructor: Mr. Mahesh Kumar Chaubey [illegible]

Mr. Ajay Kumar [illegible]

Course Leader: Mr. Mahesh Kumar Chaubey [illegible]

Forwarded by: HOD | *Approved By: Director(I/C)*

(Dr. A.K. Srivastava) | **(Dr. Vikas Nath)**

Bharati Vidyapeeth (Deemed to be University)
Institute of Management & Research, New Delhi
An ISO 9001: 2008 & 14001:2004 Certified Institute
A-4, Paschim Vihar, New Delhi-110063(Ph:011-25284396,25285808,
Fax No. 011-25286442)

Note: "Strictly for Internal academic use only"

Table of Content

Bharati Vidyapeeth Deemed University Institute of Management and Research, New Delhi

Course BCA–III SEM; ***Academic Year 2020-21***
Course Title: DBMS-II
Course Credit: 4 ***Course code: 303*** ***Credit Hours: 40***

Course Overview
Oracle revolutionized the field of enterprise database management systems with the release of Oracle Database 10g by introducing the industry's first self-management capabilities built right into the database kernel. Today, after several releases and continuous improvement of this intelligent management infrastructure, Oracle Database provides the most extensive self-management capabilities in the industry, ranging from zero-overhead instrumentation to integrated self-healing and business-driven management. Oracle's Database management capabilities make DBA lives easier by providing a full-lifecycle solution encompassing change and configuration management, patching, provisioning, testing, masking/sub setting, performance management and automatic tuning. Oracle Increase DBA productivity by 80% and reduce database testing time by 90%.

The course includes a detailed overview of Sql and PL/SQL language to which help the programmer to manage the database effectively. The objective of this subject is to provide theoretical as well as practical implementation of database management system with oracle.

Objective: The main objective this course is to teach the concepts related to database techniques and operations. SQL is introduced in this subject which helps students to build strong foundation for application of data design.
Prerequisite: knowledge of database and RDBMS technology
Course Outcomes After undergoing this course, the participants will be able to:

CO1: Understand basic database concept.

CO2: Write and Execute Simple Query using sample datasets

CO3: Write and Execute Complex queries using SQL

CO4: write and Execute PL/SQL blocks

List of Topics/modules:

Topics/modules:	**Contents/ concepts**
Module I: : Overview of l Data base Management system SQL and Components of SQL:	Introduction to Oracle: History, Features, Versions of Oracle,Spool command. Defining a database in SQL, Components of SQL: DDL, DML, DCL, DQL, SQL query Rules, Data types, Keywords, Delimiters, Literals. DDL Commands – Defining a database in SQL, Creating table, changing table definition, removing table. Truncating Table. DML Commands- Inserting, updating, deleting data, DQL Commands: Select Statement with all options. Renaming table, Describe Command, Distinct Clause, Sorting Data in a Table, Creating table from a table, Inserting data from other table, Table alias, and Column alias. **Data Constraints**: Primary key, Foreign Key, NOT NULL, UNIQUE, CHECK constraint
Module II Operators:	Arithmetic, Logical, Relational, Range Searching, Pattern Matching, IN & NOT IN Predicate, all, % any, exists, not exists clauses, Set Operations: Union, Union All, Minus, Intersect.
Module III: Joins and Oracle Functions:	Relating data through join concept. Simple join, equi join, non equi join, Self-join, Outer join, Subqueries, Aggregate Functions, Numeric Functions, String Functions, Conversion functions, Date conversion functions, Date functions.
Module IV: Database Objects:	Index: Creating index, simple index, composite index, unique index, dropping indexes, multiple indexes on table, using row-id to delete duplicate rows from a table, Sequence: Creating sequence, altering sequence, dropping sequence. Views: Using and defining, modifying, deleting. Insert / Drop / filter view. Complex view. Objects:Declaring and initializing objects in SQL,Manipulating objects inPL/SQL

Module V: Introduction to PL/SQL programming:	Introduction, Advantages, PL/SQL Block, PL/SQL Execution Environment, PL/SQL Character set, Literals, Data types, Variables, Constants, Displaying User Message on screen, Conditional Control in PL/SQL, Iterative Control Structure: While Loop, For Loop, Go to Statement.
Module VI : Advanced Programming Techniques of PL/SQL:	**Cursors**: Introduction, Types of Cursors: Implicit Cursor, Explicit Cursors, Parameterized cursors, Programs on cursors, **Triggers**: Introduction, Use of triggers, Types of Triggers, Creating triggers, Examples on Triggers, **Stored Procedures / Functions:** Introduction, How oracle executes procedures/ functions, Advantages, How to create Procedures & Functions, Examples.

Session Plan:

Session No	Detail	Learning Resources	Learning outcomes
Module I: Introduction to Relational Database Management System:			
1.	Introduction to Oracle: History, Features, Versions of Oracle,	Handouts	Will know about history, feature and version of oracle. LO1
2.	Database Structure: Logical Structure and Physical Structure,	Handouts	Will be able to differentiate logical and physical structure of database LO1
3.	Oracle Architecture: System Global Area	Handouts	Will know the working of SGA LO1
4.	Processes: Server Processes, Background Processes	Handouts	Will know that what are the processes which runs at server side and what are the background process. LO1
5.	Tools of Oracle.	Pg. 8	Will be able understand the use of various tool of oracle(server side and client side tools) LO1
6.	Spool command and its uses, how to write an output to a text file		HAndouts
7.	Defining a database in SQL, Components of SQL:	Pg.10	Will be able to know about various components of oracle LO1, LO2
8.	Components of SQL: DDL, DML, DCL, DQL,	Pg 10	Will be able to use various type of sql commands LO1, LO2

9.	SQL query Rules, Data types	Pg 10	Will be able to understand various rule which they need to keep in mind while writing sql commands LO1 &2
10.	Keywords, Delimiters, Literals, DDL Commands :Defining a database in SQL,	Pg 11	Will be able to have good idea about keywords, delimiters, literals and will be able to use them LO1 &2
11.	Creating table, changing table definition, removing table. Truncating Table.	Pg 12	Will be able to create, update table definition and will be able to drop and truncate table. LO1 &2
12.	DML Commands- Inserting, updating, deleting data,	Pg 12	Will be able to use DML commands LO1 &2
13.	DQL Commands: Select Statement with all options.	Pg. 12	Will be able to use DQL commands LO1 &2
14.	Renaming table, Describe Command,	Hand out	Will be able to rename and describe a table definition LO1 &2
15.	Distinct Clause	Pg 122	Will be able to understand and use distinct clause LO1 &2
16.	Sorting Data in a Table,	Pg 123	Will be able to sort table record on various parameters LO1 &2
17.	Creating table from a table, Inserting data from other table	Pg 124	Will be able to create table from table and insert data from other table LO1 &2
18.	Table alias, and Column alias.		Will be able to use table alias and column alias LO1 &2
19.	Data Constraints: Primary key,	Pg 137	Will understand concept of primary key and how to use at table level
20.	Foreign Key, NOT NULL, UNIQUE, CHECK constraint	Pg 138-160	Will understand and use concept of Foreign Key, NOT NULL, UNIQUE, CHECK constraintLO1 &2
Module II Operators:			
21.	Arithmetic operator, Range search operator	Pg 160,164	Will be able to use arithmetic and range search operator**LO1 &3**
22.	Logical operator, Relational operator	Pg 161	Will be able to use logical and relational operator **LO1 &3**
23.	Pattern Matching operator, IN & NOT IN Predicate,	Pg 163-165	Will be able to use pattern matching and IN and NOT IN predicate**LO1 &3**
24.	all, % any, exists, not exists clauses,	Pg 166	Will be able to use % any, exist and not

			exits**LO1 &3**
25.	Set Operations: Union, Union All, Minus, Intersect.	Pg 222	Will be able to use Union, Union All, Minus and intersect.
UnitIII: Joins and Oracle Functions:			
26.	Relating data through join concept.	**Pg. 208**	Will be able to use and understand concept of join **LO1 &3**
27.	Simple join, equi join, non equi join	**Pg 208**	Will be able to use and understand Simple join, equi join, non equi join**LO1 &3**
28.	Self-join, Outer join,	**Pg. 228**	Will be able to use and understand Self-join, Outer join,**LO1 &3**
29.	Subqueries,	**Pg 198**	Will be able to write subquery LO3
30.	Subqueries,	**Pg 198**	Will be able to write complex sub query LO3
31.	Aggregate Functions	**Pg 169**	Will be able to understand and use Aggregate Functions LO3
32.	Numeric Functions	**Pg 171**	Will be able to understand and use Numeric Functions LO3
33.	String Functions,	**Pg 174**	Will be able to understand and use string Functions LO3
34.	Conversion functions, Date conversion functions, Date functions.	**Pg 180**	Will be able to covert date data in to other types LO3
35.	,Date conversion functions, Date functions.	**Pg 180**	Will be able to use date conversion function and date function LO3
Unit IV: Database Objects:			
36.	Index: Creating index, simple index,	**Pg 239-249**	Will be able to understand use and able to create index LO2 AND3
37.	composite index, unique index, dropping indexes,	**Pg 239-249**	Will be able to understand use and able to create unique, complex index LO3
38.	multiple indexes on table	**Pg 239-249**	Will be able to understand multiple index LO3
39.	using row-id to delete duplicate rows from a table	**Pg 239-249**	Will be able to remove duplicate record using ROWID LO3
40.	Sequence: Creating sequence, altering sequence, dropping sequence	**Pg 267**	Will be able create , alter and drop sequence LO3

41.	Views: Using and defining, modifying, deleting. Insert / Drop / alter view. Complex view.	**Pg 251**	Will be able to understand concept of view, create , modify , insert data into view LO3
42.	Objects: Declaring and initializing objects in SQL, Manipulating objects in PL/SQL		
Unit V: Introduction to PL/SQL programming:			
43.	Introduction, Advantages, PL/SQL Block,	**Pg320**	Will be able to understand concept of PL/SQL programming, and its block structure LO4
44.	PL/SQL Execution Environment	**Pg 321**	Will be able to use PL/SQL environment LO4
45.	PL/SQL Character set,	**Pg 322**	Will be able to understand PL/SQL Character set,LO4
46.	Literals, Data types, Variables, Constants	**Pg 322**	Will be able to useLiterals, Data types, Variables, Constants LO4
47.	Displaying User Message on screen,	**Pg 326**	Will be able to display Displaying User Message on screen, LO4
48.	Conditional Control in PL/SQL,	**Pg 326**	Will be able to use conditional control LO4
49.	Iterative Control Structure: While Loop, For Loop, Goto Statement.	**Pg 328**	
50.	Iterative Control Structure: While Loop, For Loop, Go to Statement.	**Pg 328**	Will be able to used different looping structure LO4
51.	Iterative Control Structure: While Loop, For Loop, Go to Statement.	**Pg 328**	Will be able to write program based on loops LO4
Unit VI: Advanced Programming Techniques of PL/SQL:			
52.	Cursors: Introduction, Types of Cursors: Implicit Cursor, Explicit Cursors, Parameterized cursors, Programs on cursors,	**Pg 336-352**	Will be able to understand concept of cursor and will be able to understand difference between implicit and explicit cursor LO4
53.	Cursors: Introduction, Types of Cursors: Implicit Cursor, Explicit Cursors, Parameterized cursors, Programs on cursors,	**Pg 336-352**	Will be able to write program based on cursor, and will be able to use cursors in pl/sql code blocks LO4
54.	Triggers: Introduction, Use of triggers, Types	**Pg403-**	Will be able to understand use of

	of Triggers, Creating triggers, Examples on Triggers,	**407**	cursor and advantage of trigger, will be able to define and use cursor LO4
55.	Stored Procedures / Functions: Introduction, How oracle executes procedures/ functions, Advantages, How to create Procedures & Functions, Examples.	**Pg 379-387**	Will be able write procedure and functions LO4
56.	Stored Procedures / Functions: Introduction, How oracle executes procedures/ functions, Advantages, How to create Procedures & Functions, Examples. Stored Procedures / Functions: Introduction, How oracle executes procedures/ functions, Advantages, How to create Procedures & Functions, Exampl	**Pg 379-387**	Will be able write procedure and functions LO4
57.	Stored Procedures / Functions: Introduction, How oracle executes procedures/ functions, Advantages, How to create Procedures & Functions, Examples. Stored Procedures / Functions: Introduction, How oracle executes procedures/ functions, Advantages, How to create Procedures & Functions, Examples	**Pg 379-387**	Will be able write procedure and functions LO4

Evaluation Criteria:

SN	**CES TYPE**	**NUMBER OF CES**
1	Internal exams	**2 (10 marks each)**
2	Quiz1	**1(using Moodle) 5 marks**
3	Quiz2	**1(Using Moodle) 5 marks**
4	Class test/ Viva	**1 – 5 marks**

Recommended/ Reference Text Books and Resources:

Text Books	Ivan Bayross SQL, PL/SQL The programming language of oracle 3rd Edition BPB publication.
Reference books	1.Doing SQL from PL/SQL:Best and Worst Practices Oracle press
Internet Resource:	1. www.Orafaq.com 2. Oracle.com

9. Contact Details:

Name of the Instructor:	Mr. Mahesh Kr. Chaubey
Office Location:	BVIMR , A-4 Paschim Vihar New Delhi
Email:	maheshkumar.chaubey@bharatividyapeeth.edu, mkchaubey@gmail.com,
Website:	www.bvimr.com
Office Hours:	9:00AM-5:00PM

Study Notes

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— 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.

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, secale)	Floating type with maximum precision of 38 decimal digits
DOUBLE PRECISION	ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
FLOAT	ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
INT	ANSI specific integer type with maximum precision of 38 decimal digits
INTEGER	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
SMALLINT	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
REAL	Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration:

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

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

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

PL/SQL Character Data Types and Subtypes

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

Data Type	Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes

RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG	Variable-length character string with maximum size of 32,760 bytes
LONG RAW	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID	Physical row identifier, the address of a row in an ordinary table
UROWID	Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL Boolean Data Types

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

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

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

PL/SQL Datetime and Interval Types

The **DATE** datatype 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.
n 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
```

4 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):= 'www.bvimrcampus.com!';
BEGIN
   dbms_output.put_line(message);
END; /
```

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

```
www.bvimrcampus.com

PL/SQL procedure successfully completed.
```

5. 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: =, <, >, <=, >=, <>, !=, ~=, ^=, 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:

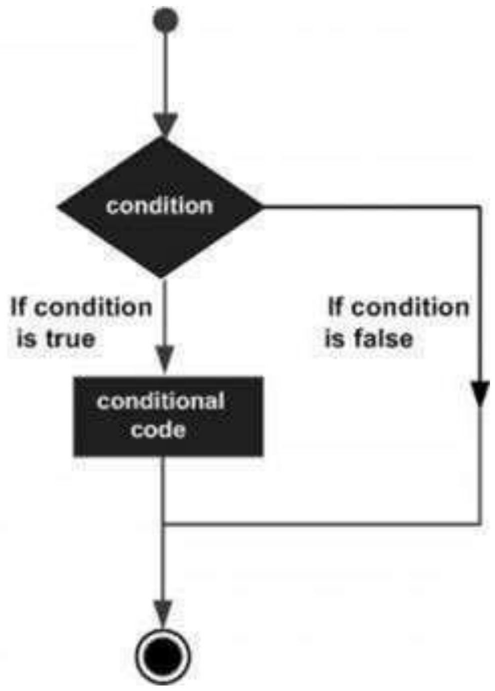
```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is  : 90
Value of (a + b) * (c / d) is  : 90
Value of a + (b * c) / d is  : 50
```

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

5 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-THENELSIF** 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:

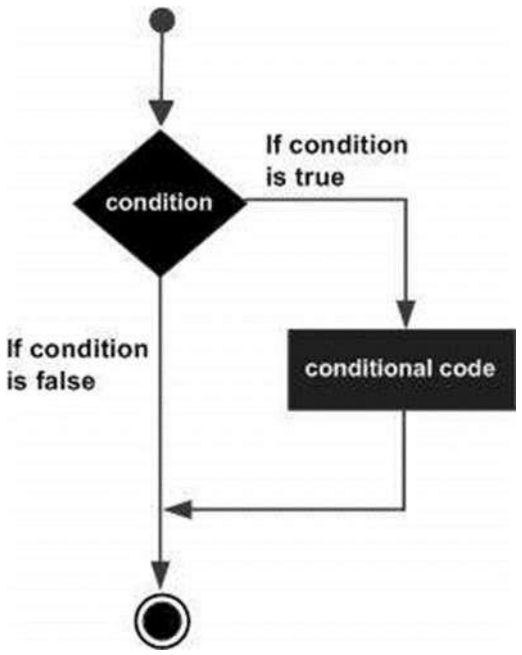
```
IF condition THEN
   S;
END IF;
```

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:

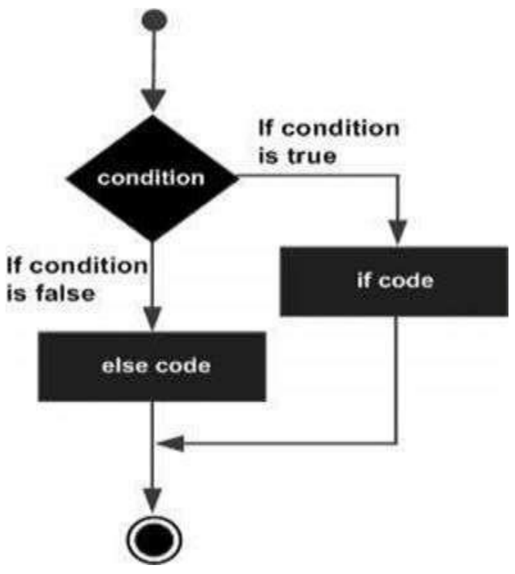
```
IF condition THEN

   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 **IFTHEN** 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

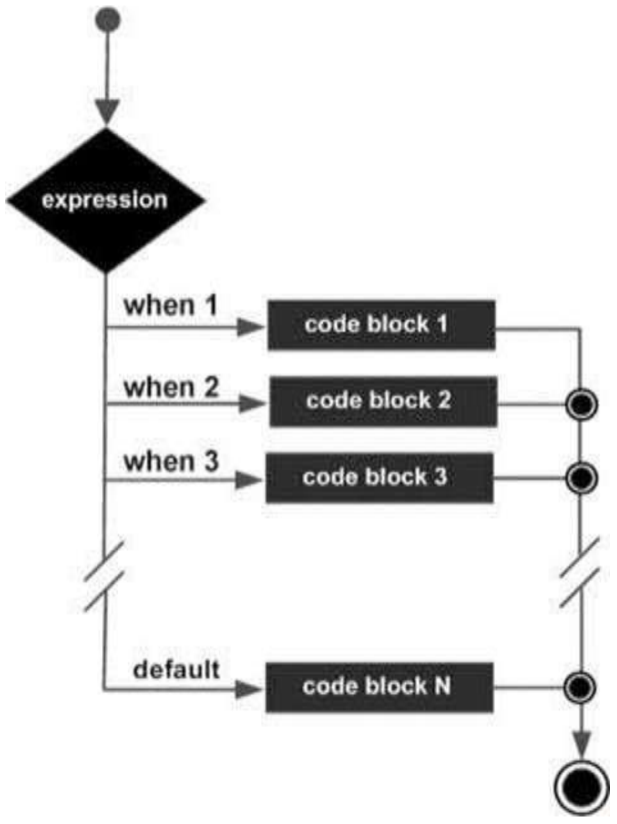
Like the **IF** statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression, the value of which is used to select one of several alternatives.

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 ch  ar(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

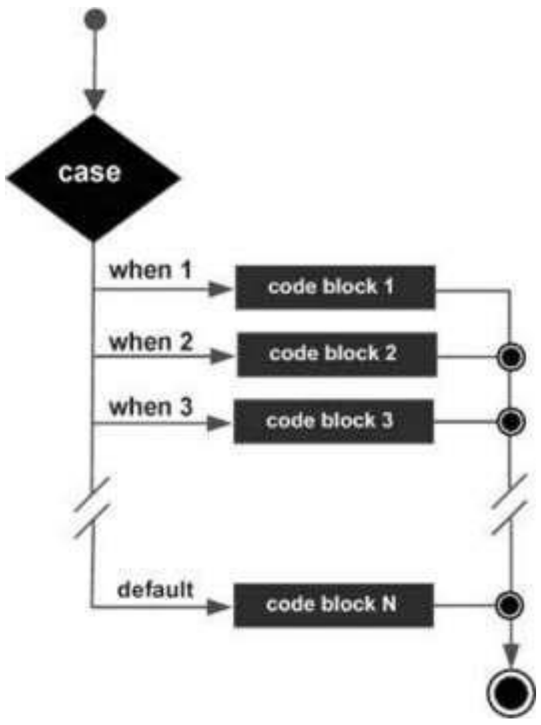
The searched **CASE** statement has no selector and the **WHEN** clauses of the statement contain search conditions that give Boolean values.

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:

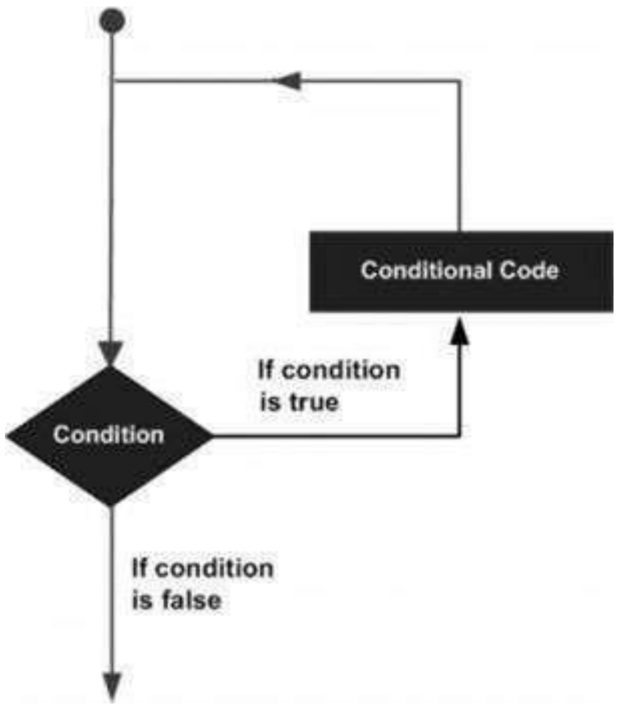
```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200


PL/SQL procedure successfully completed.
```

6 PL/SQL— Loops

6In 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:

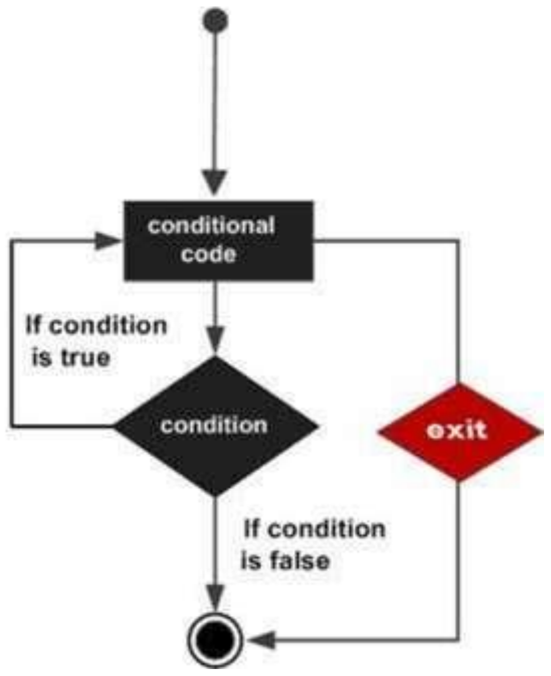
- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- If you are using nested loops (i.e., one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and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

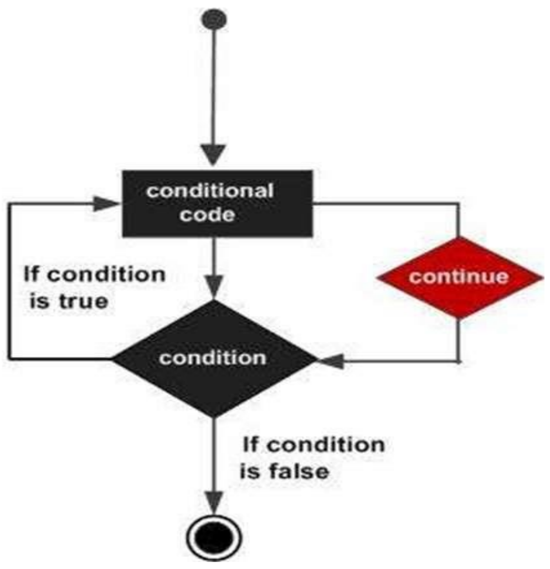
The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

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.

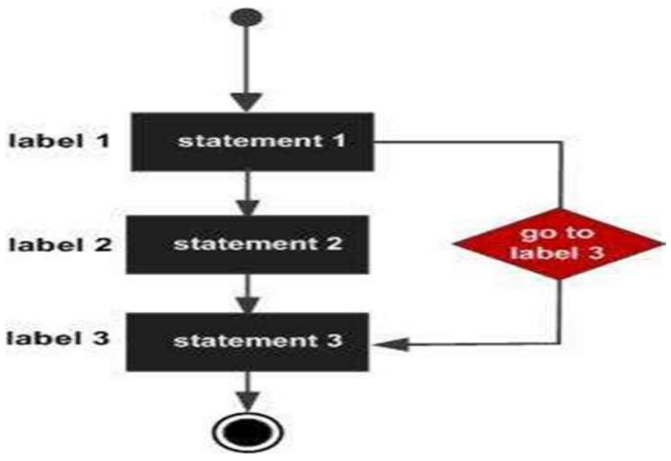
NOTE: The use of GOTO statement is not recommended in any programming language because it makes it difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

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.

7 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'
```

```
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.';
```

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:

```
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:

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.
```

8 PL/SQL— Procedures

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

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, 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);
```

9 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.
```

10 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 abovedefined 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.
```

11 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 **cursorbased** 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.
```

12 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 sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

13 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
```

14 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
```

15 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
Variablesize 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 **indexby** 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 **n**th 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 **i**th 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 **n**th 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.

16 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.
```

PL/SQL PROGRAMMS

Q1:-WRITE A PL/SQL PROGRAMM TO FIND SUM OF TWO NUMBERS AND SHOW IT?

```
Declare
a number;
b number;
c number;
Begin
a:=&a;
b:=&b;
c:=a+b;
 dbms_output.put_line ('Sum of ' || a || ' and ' || b || ' is ' || c);
End;
```

Q2:- WRITE A PL/SQL PROGRAM TO FIND SUM FIST 100 NUMBERS.

```
Declare
        a number;
        s1 number default 0;
Begin
        a:=1;
        loop
                s1:=s1+a;
                exit when (a=100);
                a:=a+1;
        end loop;
        dbms_output.put_line('Sum between 1 to 100 is '||s1);
End;
```

Q3:- WRITE A PL/SQL PROGRAM TO FIND SUM OF ODD NUMBER USING FOR LOOP …BY USER INPUT.

```
Declare
    N number;
    Sum1 number default 0;
endvalue number;
Begin
endvalue:=&endvalue;
    --n: =1;
    For n in 1..endvalue
    Loop
      If mod (n, 2) =1then
        sum1:=sum1+n;
      End if;
    End loop;
    dbms_output.put_line ('sum = ' || sum1);
End;
```

Q4:-FIND SUM OF 100 ODD NUMBERS USING WHILE LOOP .

```
Declare
n number;
endvalue number;
sum1 number default 0;
Begin
endvalue:=&endvalue;
n:=1;
    While (n <endvalue)
    Loop
      sum1:=sum1+n;
n:=n+2;
    End loop;
dbms_output.put_line ('Sum of odd numbers between 1 and ' || endvalue || ' is ' || sum1);
End;
```

Q5. WRITE A PL/SQL BLOCK TO CALCULATE NET SALARY.

```
Declare
```

```
ename varchar2(15);
       Basic number;
       DA number;
       TA number;
       IT number;
      Gross number;
       HRA number;
       PF number;
       NET_SAL number;
Begin
ename:='&ename';
       Basic: =&basic;
da:=basic * (41/100);
hra:=basic * (15/100);
       TA:= basic*(30/100);
       IT:= basic *(8/100);
       If (basic < 3000) then
           Pf: =basic * (5/100);
elsif (basic >= 3000 and basic <= 5000) then
           Pf: =basic * (7/100);
elsif (basic >= 5000 and basic <= 8000) then
           Pf: =basic * (8/100);
       Else
           Pf: =basic * (10/100);
       End if;
       Gross:=basic+TA+DA+HRA;
Net_sal:=gross-(pf+it);
       dbms_output.put_line ('Employee name: ' || ename);
       dbms_output.put_line (‘Basic salary     : ' || basic);
       dbms_output.put_line ('HRA ' || HRA);
       dbms_output.put_line (‘DA: ' || DA);
       dbms_output.put_line (‘TA: ' || TA);
       dbms_output.put_line (‘IT ' || IT);
        dbms_output.put_line (‘PF: ' || Pf);
       dbms_output.put_line (Gross Sal: ' || Gross);
       dbms_output.put_line (‘Net Salary: ' || Net_sal);
End;
```

Q6:-WRITE A PL/SQL BLOCK TO CHECK WHETHER A GIVEN NUMBER IS PRIME NO OR NOT.

```
/
DECLARE
            No NUMBER (3):= &no;
            A NUMBER (4);
            B NUMBER (2);
BEGIN
            FOR i IN 2..no - 1
LOOP
            A: = no MOD i;
            IF a = 0 THEN
                        GOTO out;
            END IF;
END LOOP;
<<Out>>
IF a = 1 THEN
DBMS_OUTPUT.PUT_LINE (no || ' is a prime number');
ELSE
DBMS_OUTPUT.PUT_LINE (no || ' is not a prime number');
END IF;
END;
```

Q7:- PL/SQL BLOCK TO SHOW LOOP.. EXIL WHEN

```
Declare
```

```
A number: = 100;
Begin
Loop
A: = a+25;
Exit when a=250;
End loop;
dbms_output.put_line (a);
End;
```

EXAMPLE OF WHILE LOOP

```
Declare
I number: =0;
J number: = 0;
Begin
While i<=100 loop
J: = j+1;
I: = I +2;
End loop;
dbms_output.put_line (I);
End;
```

GENERATE ODD NOS: FROM 1 TO 10 AND FIND ITS SUM.

```
DECLARE

I NUMBER(4);

S NUMBER(4):=0;

BEGIN

FOR I IN 1..10 LOOP

IF MOD(I,2) <> 0 THEN

S := S+I;

DBMS_OUTPUT.PUT_LINE (I);

END IF;

END LOOP;

DBMS_OUTPUT.PUT_LINE (' THE SUM OF ODD NOS FROM 1 TO 10 = ' ||S);

END;
```

P26. WRITE A PROGRAM TO PRINT ASCII TABLE :

```
DECLARE

I NUMBER;

BEGIN

FOR I IN 33..256 LOOP

DBMS_OUTPUT.PUT (( TO_CHAR (I,'000')) || ':' || CHR(I) || ' ' );

IF MOD (I, 8) = 0 THEN
```

```
DBMS_OUTPUT.PUT_LINE(' ');

END IF;

END LOOP;

END;
```

P27. Write a program to generate prime nos from 1 to 10:

```
declare

n number;

i number;

pr number;

begin

for n in 1..10 loop

pr:=1;

fori in 2 .. n/2 loop

if mod(n,i) = 0 then

pr := 0;

end if;

end loop;

ifpr = 1 then

dbms_output.put_line(n);

end if;

end loop;

end;
```

P29. Write a program to reverse the digits of the number:

```
DECLARE

N number ;

S NUMBER : = 0;

R NUMBER;

K number;

begin
```

```
N :=&N;

K := N;

LOOP

EXIT WHEN N = 0 ;

S := S * 10;

R := MOD(N,10);

S := S + R;

N := TRUNC(N/10);

end loop;

dbms_output.put_line( ' THE REVERSED DIGITS ' || ' OF ' || K || ' = ' || S);

end;
```

P31 . Write PL/SQL SCRIPT to determine the salary of employee in the EMP table. If salary >7500 give an increment of 15% , otherwise display the message NO INCREMENT . Get the empno from the user.

```
DECLARE

ENO EMP.EMPNO%TYPE;

SALARY EMP.SAL%TYPE;

BEGIN

ENO :=&ENO;

SELECT SAL INTO SALARY FROM EMP WHERE EMPNO=ENO;

IF SALARY >7500 THEN

             UPDATE EMP SET SAL =SAL+SAL* 15/100 WHERE EMPNO=ENO;

ELSE

             DBMS_OUTPUT.PUT_LINE(NO INCREMENT)

END IF;

END;
```

Make changes in the above block of code based on following inputs

```
If salary is less than 1000 increment given is 5%

     "1000 and less than 5000 increment given is 10%

    "5000 and less than 10000 increment given is 15%

Incase in salary in more than equal to 10 +000 increment given is 20%
```

SET SERVEROUTPUT ON

A bit about comments. A comment can have 2 forms i.e.
– The comment line begins with a double hyphen (–). The entire line will be treated as a comment.
– The C style comment such as /* i am a comment */

CONDITIONAL CONTROL AND ITERATIVE CONTROL AND SEQUENTIAL CONTROL.

IF and else…..
IF –Condition THEN
–Action
ELSEIF –Condition THEN
–Action
ELSE
–Action
END IF;

SIMPLE LOOP
loop
– Sequence of statements;
end loop;

the loop ends when u use EXIT WHEN statement –condition

WHILE LOOP
while –condition
loop
–sequence of statements
end loop;

FOR LOOP
FOR i in 1..10
loop
–sequence of statements
end loop;

GOTO (sequential control)

```
GOTO X;
<< X >>
```

EXAMPLES

```
--ADDITION
declare
a number;
b number;
c number;
begin
a:=&a;
b:=&b;
c:=a+b;
 dbms_output.put_line('Sum of ' || a || ' and ' || b || ' is ' || c);
end;
```

Here & is used to take user input at runtime.....

```
--SUM OF 100 NUMBERS

Declare
        a number;
        s1 number default 0;
Begin
        a:=1;
        loop
                s1:=s1+a;
                exit when (a=100);
                a:=a+1;
        end loop;
        dbms_output.put_line('Sum between 1 to 100 is '||s1);
End;
```

```
--SUM OF odd NUMBERS USING USER INPUT...for loop
declare
n number;
sum1 number default 0;
endvalue number;
begin
endvalue:=&endvalue;
n:=1;
for n in 1.. endvalue
loop
if mod(n,2)=1
then
        sum1:=sum1+n;
```

```
end if
end loop;
      dbms_output.put_line('sum = ' || sum1);
end;
```

```
--SUM OF 100 ODD NUMBER .. WHILE LOOP
declare
n number;
endvalue number;
sum1 number default 0;
begin
endvalue:=&endvalue;
n:=1;
while (n <endvalue)
loop
          sum1:=sum1+n;
n:=n+2;
end loop;
dbms_output.put_line('Sum of odd numbers between 1 and ' || endvalue || ' is ' ||
sum1);
end;
```

```
--CALCULATION OF NET SALARY
declare
ename varchar2(15);
basic number;
da number;
hra number;
pf number;
netsalary number;
begin
ename:=&ename;
basic:=&basic;

da:=basic * (41/100);
hra:=basic * (15/100);

if (basic < 3000)then
pf:=basic * (5/100);
elsif (basic >= 3000 and basic <= 5000) then
pf:=basic * (7/100);
elsif (basic >= 5000 and basic <= 8000) then
pf:=basic * (8/100);
else
pf:=basic * (10/100);
end if;
netsalary:=basic + da + hra -pf;
      dbms_output.put_line('Employee name : ' || ename);
dbms_output.put_line('Providend Fund : ' || pf);
```

```
        dbms_output.put_line('Net salary : ' || netsalary);
end;
```

```
--MAXIMUM OF 3 NUMBERS
Declare
                a number;
                b number;
                c number;
                d number;
Begin
                dbms_output.put_line('Enter a:');
                a:=&a;
                dbms_output.put_line('Enter b:');
                b:=&b;
                dbms_output.put_line('Enter c:');
                c:=&b;
                if (a>b) and (a>c) then
                                dbms_output.putline('A is Maximum');
                elsif (b>a) and (b>c) then
                                dbms_output.putline('B is Maximum');
                else
                                dbms_output.putline('C is Maximum');
                end if;
End;
```

```
--QUERY EXAMPLE--IS SMITH EARNING ENOUGH
declare
        s1 emp.sal %type;
begin
selectsal into s1 from emp
whereename = 'SMITH';
if(no_data_found) then
raise_application_error(20001,'smith is not present');
end if;

if(s1 > 10000)then
raise_application_error(20002,'smith is earning enough');
end if;

updateemp set sal=sal + 500
whereename='SMITH';
end;
```

```
--PRIME NO OR NOT
DECLARE
                no NUMBER (3) := &no;
                a NUMBER (4);
                b NUMBER (2);
```

```
BEGIN
        FOR i IN 2..no - 1
        LOOP
                a := no MOD i;
                IF a = 0 THEN
                        GOTO out;
                END IF;
        END LOOP;
        <>
        IF a = 1 THEN
        DBMS_OUTPUT.PUT_LINE (no || ' is a prime number');
        ELSE
        DBMS_OUTPUT.PUT_LINE (no || ' is not a prime number');
        END IF;
END;
```

```
--SIMPLE EXAMPLE OF LOOP STATEMENT I.E. EXIT WHEN
Declare
a number:= 100;
begin
loop
a := a+25;
exit when a=250;
end loop;
dbms_output.put_line (to_Char(a));
end;
```

```
--EXAMPLE OF WHILE LOOP
Declare
i number:=0;
j number:= 0;
begin
whilei<=100 loop
j := j+1;
i := i +2;
end loop;
dbms_output.put_line (to_char(i));
end;
```

```
--EXAMPLE OF FOR LOOP
Declare
Begin
For i in 1..10
Loop
dbms_output.put_line(to_char(i));
End loop;
end;
```

```
--SEQUENTIAL CONTROL GOTO.....
```

```
declare
--takes the default datatype of the column of the table price
costprice.minprice%type;
begin
selectstdprice into cost from price where prodial in (Select prodid from product
where prodese = "shampoo");
if cost > 7000 then
gotoUpd;
end if;
<<Upd>>
Update price set minprice = 6999 where prodid=111;
end;
```

```
--CALCULATE THE AREA OF A CIRCLE FOR A VALUE OF RADIUS VARYING FROM 3
TO 7. STORE THE RADIUS AND THE CORRESPONDING VALUES OF CALCULATED
AREA IN A TABLE AREAS.

Declare
pi constant number(4,2) := 3.14;
radius number(5);
area number(14,2);

Begin
radius := 3;
While radius <=7
Loop
area := pi* power(radius,2);
Insert into areas values (radius, area);
radius:= radius+1;
end loop;
end;
```

```
--INVERTING A NUMBER 5639 TO 9365
Declare
given_numbervarchar(5) := '5639';
str_length number(2);
inverted_numbervarchar(5);

Begin
str_length := length(given_number);
For cntr in reverse 1..str_length
loop
inverted_number := inverted_number || substr(given_number, cntr, 1);
end loop;
dbms_output.put_line('The Given no is ' || given_number);
dbms_output.put_line('The inverted number is ' || inverted_number);
end;
```

EXCEPTION HANDLING

Errors in pl/sql block can be handled...error handling refers to the way we handle the errors in pl/sql block so that no
crashing stuff of code takes place...This is exactly the same as we do in C++ or java..right!!
There are two type:
===> predefined exceptions
===>user defined exceptions
The above 2 terms are self explanatory

predefined exceptions:

No-data-found	==	when no rows are returned
Cursor-already-open	==	when a cursor is opened in advance
Dup-val-On-index	==	for duplicate entry of index..
Storage-error	==	if memory is damaged
Program-error	==	internal problem in pl/sql
Zero-divide	==	divide by zero
invalid-cursor	==	if a cursor is not open and u r trying to close it
Login-denied	==	invalid user name or password
Invalid-number	==	if u r inserting a string datatype for a number datatype which is already declared
Too-many-rows	==	if more rows r returned by select statement

SYNTAX

```
begin
sequence of statements;
exception
when --exception name then
sequence of statements;
end;
```

EXAMPLES
--When there is no data returned by row

```
declare
priceitem.actualprice%type;
begin
```

```
Select actual price into price from item where qty=888;
when no-data-found then
dbms_output.put_line('item missing');
end;
```

```
--EXAMPLE OF USER DEFINED EXCEPTION
DECLARE
e_recemp%ROWTYPE;
e1 EXCEPTION;
 sal1 emp.sal%TYPE;
BEGIN
 SELECT sal INTO sal1 FROM emp WHERE deptno = 30 AND ename = 'John';
 IF sal1 < 5000 THEN
  RAISE e1;
sal1 := 8500;
  UPDATE emp SET sal = sal1 WHERE deptno = 30 AND ename = 'John';
 END IF;
 EXCEPTION
  WHEN no_data_found THEN
   RAISE_APPLICATION_ERROR (-20001, 'John is not there.');
  WHEN e1 THEN
   RAISE_APPLICATION_ERROR (-20002, 'Less Salary.');
END;
```

EXAMPLE OF RAISE-APPLICATION-ERROR... THIS IS YOUR OWN ERROR STATEMENT...U RAISE
--YOUR OWN ERROR

```
Declare
s1 emp.sal %type;
begin
selectsal into s1 from emp where ename='SOMDUTT';
if(no-data-found) then
raise_application_error(20001, 'somdutt is not there');
end if;
if(s1 > 10000) then
```

```
raise_application_error(20002, 'somdutt is earing a lot');
end if;
updateemp set sal=sal+500 where ename='SOMDUTT';
end;
```

```
--INTERESTING EG OF USER DEFINED EXCEPTIONS
Declare
zero-price exception;
price number(8);
begin
selectactualprice into price from item where ordid =400;
if price=0 or price is null then
raise zero-price;
end if;
exception
when zero-price then
dbms_output.put_line('raised xero-price exception');
end;
```

CURSORS

Cursor is a work area in pl/sql which is used by sql server used to store the result of a query. Each column value is pointed using pointer. You can independently manipulate cursor values. A bit about it's working….. suppose u ask for a query stored in the server … at first a cursor consisting of query result is created in server…now the cursor is transferred to the client where again cursor is created and hence the result is displayed……

Cursors are of 2 types:

implicit and explicit…….implicit cursors are created by oracle engine itself while explicit cursors are created by the users……cursors are generally used in such a case when a query returns more than one rows….normal pl/sql returning more than one rows givens error but using cursor this limitation can be avoided….so cursors are used….

Cursor attributes

%ISOPEN   == returns true if ursor is open, false otherwise

%FOUND   == returns true if recod was fetched successfully, false otherwise

%NOTFOUND == returns true if record was not fetched successfully, false otherwise

%ROWCOUNT == returns number of records processed from the cursor.

Very important: Cursor can be controlled using following 3 control statements. They are Open, Fetch, Close…..open statement identifies the active set…i.e. query returned by select statement…close statement closes the cursor…and fetch statement fetches rows into the variables…Cursors can be made into use using cursor for loop and fetch statement…we will see the corresponding examples…

EXAMPLES

```
--EXAMPLE OF SQL%FOUND (IMPLICIT CURSORS)
begin
update employee set salary=salary *0.15
whereemp_code = &emp_code;
ifsql%found then
dbms_output.put_line('employee record modified successfully');
else
dbms_output.put_line('employee no does not exist');
end if;
end;
```

```
--EXAMPLE FOR SQL%NOTFOUND (IMPLICIT CURSORS)
begin
update employee set salary = salary*0.15 where emp_code = &emp_code;
ifsql%notfound then
dbms_output.put_line('employee no . does not exist');
else
dbms_output.put_line('employee record modified successfully');
end if;
end;
```

```
--EXAMPLE FOR SQL%ROWCOUNT (IMPLICIT CURSORS)
```

```
declare
rows_affected char(4);
begin
update employee set salary = salary*0.15 where job='programmers';
rows_affected := to_char(sql%rowcount);
ifsql%rowcount> 0 then
dbms_output.put_line(rows_affected || 'employee records modified successfully');
else
dbms_output.put_line('There are no employees working as programmers');
end if;
end;
```

Syntax of explicit cursor: Cursor cursorname is sql select statement;

Syntax of fetch : fetch cursorname into variable1, variable2…;

Syntax of close; close cursorname;

Syntax of open cursor; open cursorname;

```
--EXPLICIT CURSOR EG
DECLARE
        CURSOR c1 is SELECT * FROM emp;
        str_empnoemp.empno%type;
        str_enameemp.ename%type;
        str_jobemp.job%type;
        str_mgremp.mgr%type;
        str_hiredateemp.hiredate%type;
        str_salemp.sal%type;
        str_commemp.comm%type;
        str_deptnoemp.deptno%type;
        rno number;
BEGIN
        rno := &rno;
        FOR e_rec IN c1
```

```
          LOOP
                    IF c1%rowcount = rno THEN
                              DBMS_OUTPUT.PUT_LINE (str_empno || ' ' || str_ename || ' ' ||
str_job || ' ' || str_mgr || ' ' || str_hiredate || ' ' || str_sal || ' ' || str_comm || ' ' || str_deptno);
                    END IF;
          END LOOP;
END;
```

--ANOTHER DISPLAYING VALUE OF A TABLE

```
DECLARE
          CURSOR c1 IS SELECT  * FROM emp;
          e_recemp%rowtype;
BEGIN
          OPEN c1;
          LOOP
                    FETCH c1 INTO e_rec;
DBMS_OUTPUT.PUT_LINE('Number: ' || ' ' || e_rec.empno);
DBMS_OUTPUT.PUT_LINE('Name  : ' || ' ' || e_rec.ename);
DBMS_OUTPUT.PUT_LINE('Salary: ' || ' ' || e_rec.sal);
          EXIT WHEN c1%NOTFOUND;
          END LOOP;
          CLOSE c1;END;
```

-- Display details of Highest 10 salary paid employee

```
DECLARE
          CURSOR c1 IS SELECT  * FROM emp ORDER BY sal DESC;
          e_recemp%rowtype;
BEGIN
          FOR e_rec IN c1
          LOOP
DBMS_OUTPUT.PUT_LINE('Number: ' || ' ' || e_rec.empno);
DBMS_OUTPUT.PUT_LINE('Name  : ' || ' ' || e_rec.ename);
DBMS_OUTPUT.PUT_LINE('Salary: ' || ' ' || e_rec.sal);
          EXIT WHEN c1%ROWCOUNT >= 10;
```

```
        END LOOP;
END;
```

-- EXAMPLE OF CURSOR FOR LOOP

```
declare cursor c1 is select * from somdutt;
begin
foroutvariable in c1
loop
exit when c1%notfound;
ifoutvariable.age< 21 then
dbms_output.put_line(outvariable.age || ' ' || outvariable.name);
end if;
end loop;
end;
```

--ref STRONG CURSORS

```
DECLARE
        TYPE ecursor IS REF CURSOR RETURN emp%ROWTYPE;
        ecurecursor;
        e_recemp%ROWTYPE;
        dn NUMBER;
BEGIN
        dn := &deptno;
        OPEN ecur FOR SELECT * FROM emp WHERE deptno = dn;
        FOR e_rec IN ecur
        LOOP
                DBMS_OUTPUT.PUT_LINE ('Employee No    : ' || e_rec.empno);
                DBMS_OUTPUT.PUT_LINE ('Employee Salary: ' || e_rec.salary);
        END LOOP;
END;
```

```
--REF WEAK CURSORS
DECLARE
```

```
	TYPE tcursor IS REF CURSOR;
	tcurtcursor;
	e1emp%ROWTYPE;
	d1dept%ROWTYPE;
	tname VARCHAR2(20);
BEGIN
	tname := &tablename;
	IF tname = 'emp' THEN
		OPEN tcur FOR SELECT * FORM emp;
		DBMS_OUTPUT.PUT_LINE ('Emp table opened.');
		closetcur;
		DBMS_OUTPUT.PUT_LINE ('Emp table closed.');
	ELSE IF tname = 'dept' THEN
		OPEN tcur FOR SELECT * FROM dept;
		DBMS_OUTPUT.PUT_LINE ('Dept table opened.');
		closetcur;
		DBMS_OUTPUT.PUT_LINE ('Emp table closed.');
	ELSE
		RAISE_APPLICATION_ERROR (-20004, 'Table name is wrong');
	END IF;
END;
```

```
--CURSOR FOR LOOP WITH PARAMETERS
Declare
Cursor c1(Dno number) is select * from emp where deptno = dno;
begin
forempree in c1(10) loop;
dbms_output.put_line(empree.ename);
end loop;
end;
```

TRIGGERS

Trigger is a stored procedure which is called implicitly by oracle engine whenever a insert, update or delete statement is fired.

Advantages of database triggers:

—> Data is generated on it's own

—> Replicate table can be maintained

—>To enforce complex integrity contraints

—>To edit data modifications

—>Toautoincrement a field

etc..

Syntax: Create or replace trigger –triggername– [before/after] [insert/pdate/delete] on –tablename– [for each satement/ for each row] [when --condition--] plus..begin.and exception

Triggers are of following type: before or after trigger ….and for each row and for each statement trigger… before trigger is fired before insert/update/delete statement while after trigger is fired after insert/update/delete statement…for each row and for each statements triggers are self explainatory**..**

EXAMPLE

— A database trigger that allows changes to employee table only during the business hours(i.e. from 8 a.m to 5.00 p.m.) from monday to saturday. There is no restriction on viewing data from the table –

```
CREATE OR REPLACE TRIGGER Time_Check BEFORE INSERT OR UPDATE OR

DELETE ON EMP

BEGIN
```

```
IF TO_NUMBER(TO_CHAR(SYSDATE,’hh24′)) < 10 OR
TO_NUMBER(TO_CHAR(SYSDATE,’hh24′)) >= 17 OR

TO_CHAR(SYSDATE,’DAY’) = ‘SAT’ OR TO_CHAR(SYSDATE,’DAY’) = ‘SAT’ THEN
RAISE_APPLICATION_ERROR (-20004,’YOU CAN ACCESS ONLY BETWEEN 10 AM TO 5
PM ON MONDAY TO FRIDAY ONLY.’);

END IF;

END;
```

–YOU HAVE 2 TABLES WITH THE SAME STRUCTURE. IF U DELETE A RECORD FROM ONE TABLE, IT WILL BE INSERTED IN 2ND TABLE ED TRIGGERNAME

```
Create or replace trigger backup after delete on emp for each row
begin
insert into emp values (:old.ename,:old.job,:old.sal);
end;
```

save the file.. and then sql> @ triggername

–To STICK IN SAL FIELD BY TRIGGER MEANS WHEN U ENTER GREATER THAN 5000, THEN THIS TRIGGER IS EXECUTED

```
Create or replace trigger check before insert on emp for each row when (New.sal>
5000);
begin
raise_application_error(-20000, ‘your no is greater than 5000′);
end;
```

–NO CHANGES CAN BE DONE ON A PARTICULAR TABLE ON SUNDAY AND SATURDAY

```
Create or replace trigger change before on emp for each row when (to_char(sysdate,’dy’)
in (’SAT’,'SUN’))
begin
raise_application_error(-200001, ‘u cannot enter data in saturnday and sunday’);
end;
```

–IF U ENTER IN EMP TABLE ENAME FIELD’S DATA IN ANY CASE IT WILL BE INSERTED IN CAPITAL LETTERS’S ONLY

```
Create or replace trigger cap before insert on emp for each row
begin
:New.ename = upper(:New.ename);
end;
```

–A TRIGGER WHICH WILL NOT ALLOW U TO ENTER DUPLICATE VALUES IN FIELD EMPNO IN EMP TABLE

```
Create or replace trigger dubb before insert on emp for each row

Declare cursor c1 is select * from emp; x emp%rowtype;

Begin

open c1; loop fetch c1 into x; if :New.empno = x.empno then dbms_output.put_line(’you

entered duplicated no’);

elseif :New.empno is null then dbms_output.put_line(’you empno is null’);

end if;

exit when c1%notfound;

end loop;

close c1;

end;
```

Remember trigger can be dropped using Drop Trigger trigger name; statement…

PROCEDURES AND FUNCTIONS

procedure is a subprogram…which consists of a set of sql statement. Procedures are not very different from functions. A procedure or function is a logically grouped set of SQL and PL/SQL statements that perform a specific task. A stored procedure or function is a named pl/sql code block that have been compiled and stored in one of the oracle engines’ system tables.

To make a procedure or function dynamic either of them can be passed parameters before execution. A procedure or function can then change the way it works depending upon the parameters passed prior to its execution.

Procedures and function are made up of a

Declarative part,

An executable part

And an optional exception-handling part

A declaration part consists of declarations of variables. A executable part consists of the logic i.e. sql statements….and exception handling part handles any error during run-time

The oracle engine performs the following steps to execute a procedure or funtion….

1. Verifies user access,
2. Verifies procedure or function validity
3. and executes the procedure or function.

Some of the advantages of using procedures and functions are:

1. Security,
2. Performance,
3. Memory allocation,
4. Productivity, integrity.

Most important the difference between procedures and functions: A function must return a value back to the caller. A function can return only one value to the calling pl/sql block.

By defining multiple out parameters in a procedure, multiple values can be passed to the caller. The out variable being global by nature, its value is accessible by any pl/sql code block including the calling pl/sql block.

Syntax for stored procedure:

CREATE OR REPLACE PROCEDURE [schema] procedurename (argument { IN, OUT, IN OUT} data type, ..) {IS, AS}

variable declarations; constant declarations;

BEGIN

pl/sql subprogram body;

EXCEPTION

exceptionpl/sql block;

END;

Syntax for stored function:

CREATE OR REPLACE FUNCTION[schema] functionname(argument IN data type, ..)

RETURN data type {IS, AS}

variable declarations; constant declarations; BEGIN pl/sql subprogram body;

EXCEPTION

exceptionpl/sql block;

END;

The above syntax i think is self-explanatory…but i will give u some details…IN: specifies that a value for the argument must be specified when calling the procedure or function. Argument: is the name of an argument to the procedure or function. Parentheses can be omitted if no arguments are present. OUT: specifies that the procedure passes a value for this argument back to its calling environment after execution. IN OUT: specifies that a value for the argument must be specified when calling the procedure and that the procedure passes a value for this argument back to its calling environment after execution. By default it takes IN. Data type: is the data type of an argument.

EXAMPLES

--PROCEDURE USING NO ARGUMENT..AND USING CURSOR

```
CREATE OR REPLACE PROCEDURE P2 IS
cursor cur1 is select * from emp;
begin
forerec in cur1
loop
dbms_output.put_line(erec.ename);
end loop;
end;
```

--PROCEDURE USING ARGUMENT

```
CREATE OR REPLACE PROCEDURE ME(X IN NUMBER) IS
BEGIN
dbms_output.put_line(x*x);
end;

sql> exec me(3);
```

--FUNCTION using argument

```
CREATE OR REPLACE FUNCTION RMT(X IN NUMBER) RETURN NUMBER IS
```

```
BEGIN
dbms_output.put_line(x*x);
--return (x*x);
End;
```

(Make a block like this to run it.....)

```
Begin
dbms_output.put_line(rmt(3));
End;
```

--CREATE A PROCEDURE THAT DELETE ROWS FROM ENQUIRY
--WHICH ARE 1 YRS BEFORE

```
Create or replace procedure myprocedure is begin
Delete from enquiry where enquiry date <= sysdate - 1;
End;
```

--CREATE A PROCEDURE THAT TAKES ARGUMENT STUDENT NAME,
--AND FIND OUT FEES PAID BY THAT STUDENT

```
CREATE or REPLACE procedure me (name in varchar) is
cursor c1 is select a.feespaiddate from feespaid a, enrollment b, enquiry c
where
c.enquiryno = b.enquiryno and
a.rollno = b.rollno and
c.fname = namee;
begin
forerec in c1
loop
dbms_output.put_line(erec.feespaiddate);
end loop;
end;
```

--SUM OF 2 NOS

```
CREATE or replace procedure p1 is

Declare

a number;

b number;

c number;

Begin

a:=50;

b:=89;
```

```
c:=a+b;
 dbms_output.put_line('Sum of '||a||' and '||b||' is '||c);
End;
```

--DELETION PROCEDURE

```
create or replace procedure myproc is
begin
delete from enquiry where fname='somdutt';
end;
```

--IN and OUT procedure example

```
Create or replace procedure lest ( a number, b out number) is
identify number;
begin
selectordid into identity from item where
itemid = a;
if identity < 1000 then
b := 100;
end if;
end l
```

--in out parameter eg

```
Create or replace procedure sample ( a in number, b in out number) is
identity number;
begin
selectordid, prodid into identity, b from item where itemid=a;
if b<600 then
b := b + 100;
end if;
end;
```

now procedure is called by passing parameter

```
declare
a number;
b number;
```

```
begin
sample(3000, b)
dbms_output.put_line(1th value of b is 11 b);
end ;
```

```
--SIMILAR EG AS BEFORE
create or replace procedure getsal( sal1 in out number) is
begin
selectsal into sal1 from emp
whereempno =  sal1;
end ;
```

now use the above in plsql block

```
declare
sal1 number := 7999;
begin
getsal(sal1);
dbms_output.put_line('The employee salary is' || sal1);
end ;
```

U can make a procedure and functions similarly…..also if u want to drop a function then use drop function function name and for procedure use drop procedure procedure name

PACKAGES

A package is an oracle object, which holds other objects within it. Objects commonly held within a package are procedures, functions, variables, constants, cursors and exceptions. Packages in plsql is very much similar to those packages which we use in JAVA……yeah!! java packages holds numerous classes..right!!!…

A package has 2 parts…..package specification and package body

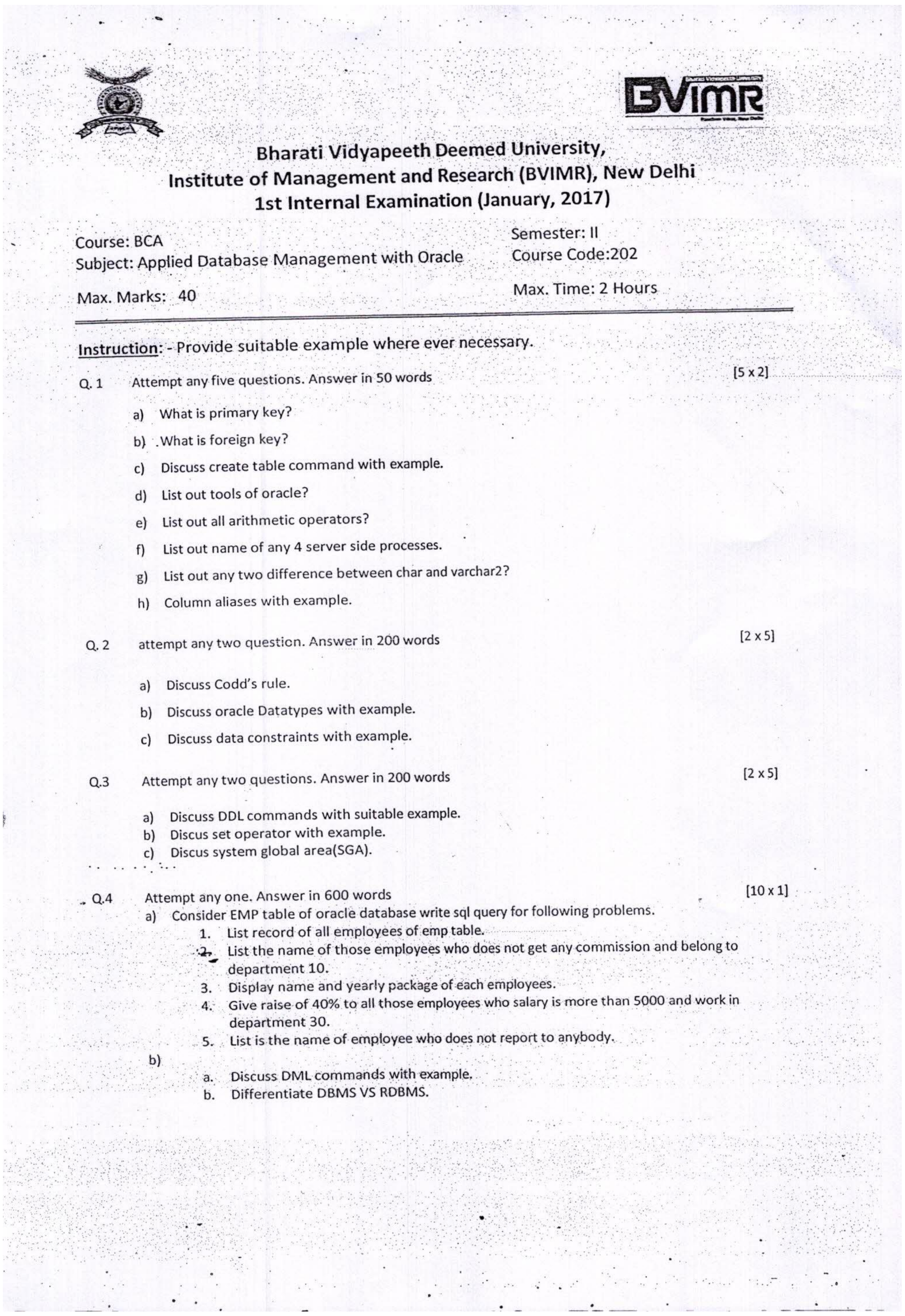
A package specification part consists of all sort of declaration of functions and procedures while package body consists of codings and logic of declared functions and procedures…

EXAMPLE

```
--SIMPLEST EG
--specification
create or replace package pack2 is
function rmt(x in number) return number;
procedure rmt1(x in number);
end;
--body
create or replace package body pack2 is
function rmt(x in number) return number is
begin
return (x*x);
end;
procedure rmt1(x in number) is
begin
dbms_output.put_line(x*x);
end;
end;
(how to run.....)
execpackagename.procedurename
i.e.
exec pack2.rmt1(3);
```

As shown above u can put in complicated procedures and functions inside the package…I have just shown a simple example…u can easily modify the above code to fit your requirement……Just try out packages which includes cursors, procedures and functions..etc..Remember pl/sql supports overloading…i.e. u can use the same function or procedure name in your application but with different no or type of arguments..

Internal Question Papers



Bharati Vidyapeeth Deemed University,
Institute of Management and Research (BVIMR), New Delhi
1st Internal Examination (January, 2017)

Course: BCA
Subject: Applied Database Management with Oracle
Max. Marks: 40

Semester: II
Course Code:202
Max. Time: 2 Hours

Instruction: - Provide suitable example where ever necessary.

Q. 1 Attempt any five questions. Answer in 50 words [5 x 2]

a) What is primary key?
b) What is foreign key?
c) Discuss create table command with example.
d) List out tools of oracle?
e) List out all arithmetic operators?
f) List out name of any 4 server side processes.
g) List out any two difference between char and varchar2?
h) Column aliases with example.

Q. 2 attempt any two question. Answer in 200 words [2 x 5]

a) Discuss Codd's rule.
b) Discuss oracle Datatypes with example.
c) Discuss data constraints with example.

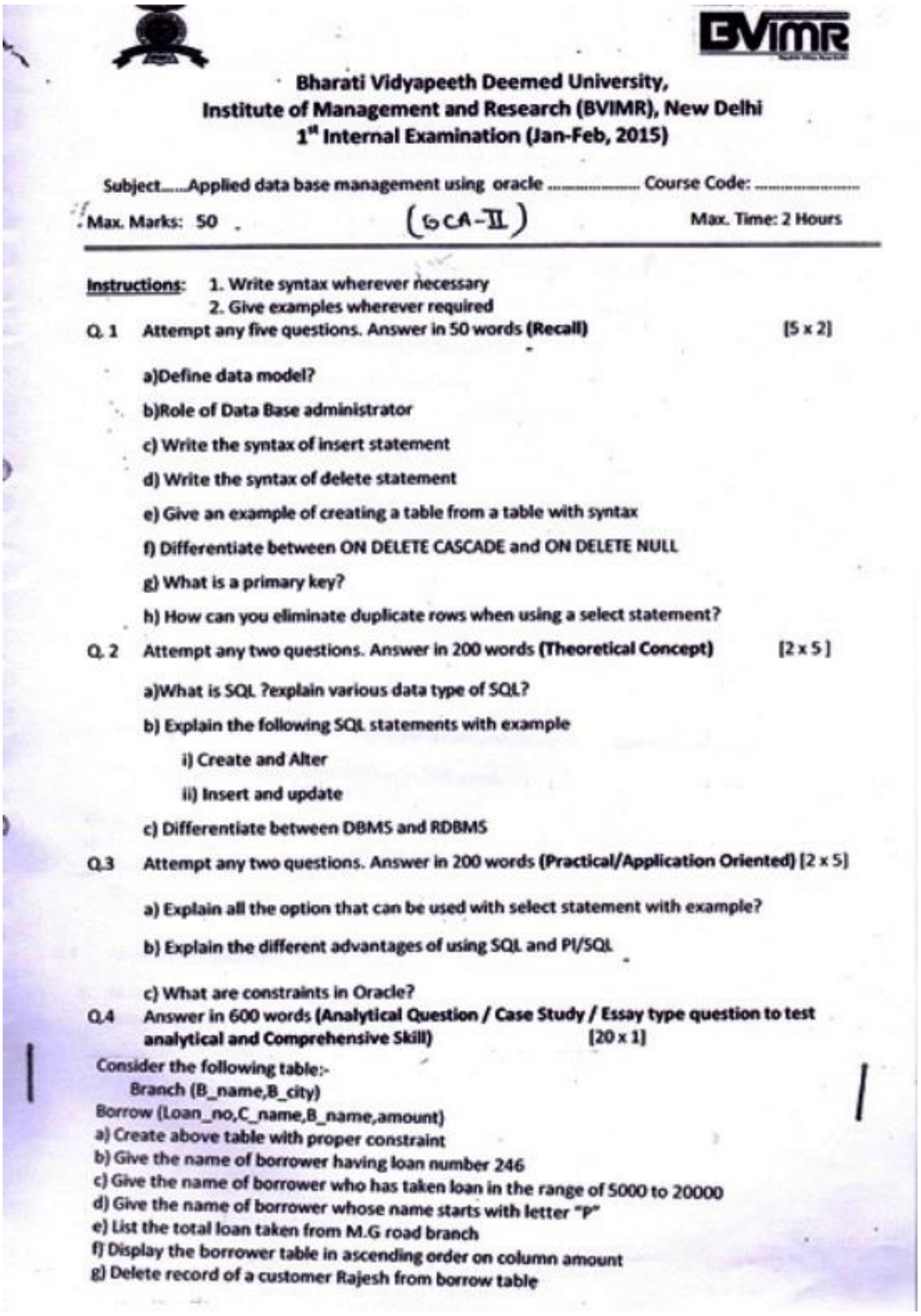
Q.3 Attempt any two questions. Answer in 200 words [2 x 5]

a) Discuss DDL commands with suitable example.
b) Discus set operator with example.
c) Discus system global area(SGA).

Q.4 Attempt any one. Answer in 600 words [10 x 1]

a) Consider EMP table of oracle database write sql query for following problems.
1. List record of all employees of emp table.
2. List the name of those employees who does not get any commission and belong to department 10.
3. Display name and yearly package of each employees.
4. Give raise of 40% to all those employees who salary is more than 5000 and work in department 30.
5. List is the name of employee who does not report to anybody.

b)
a. Discuss DML commands with example.
b. Differentiate DBMS VS RDBMS.



Bharati Vidyapeeth Deemed University,
Institute of Management and Research (BVIMR), New Delhi
1st Internal Examination (Jan-Feb, 2015)

Subject.....Applied data base management using oracle .... Course Code: ....

Max. Marks: 50 (GCA-II) Max. Time: 2 Hours

Instructions: 1. Write syntax wherever necessary
2. Give examples wherever required

Q. 1 Attempt any five questions. Answer in 50 words **(Recall)** [5 x 2]

a)Define data model?

b)Role of Data Base administrator

c) Write the syntax of insert statement

d) Write the syntax of delete statement

e) Give an example of creating a table from a table with syntax

f) Differentiate between ON DELETE CASCADE and ON DELETE NULL

g) What is a primary key?

h) How can you eliminate duplicate rows when using a select statement?

Q. 2 Attempt any two questions. Answer in 200 words **(Theoretical Concept)** [2 x 5]

a)What is SQL ?explain various data type of SQL?

b) Explain the following SQL statements with example

i) Create and Alter

ii) Insert and update

c) Differentiate between DBMS and RDBMS

Q.3 Attempt any two questions. Answer in 200 words **(Practical/Application Oriented)** [2 x 5]

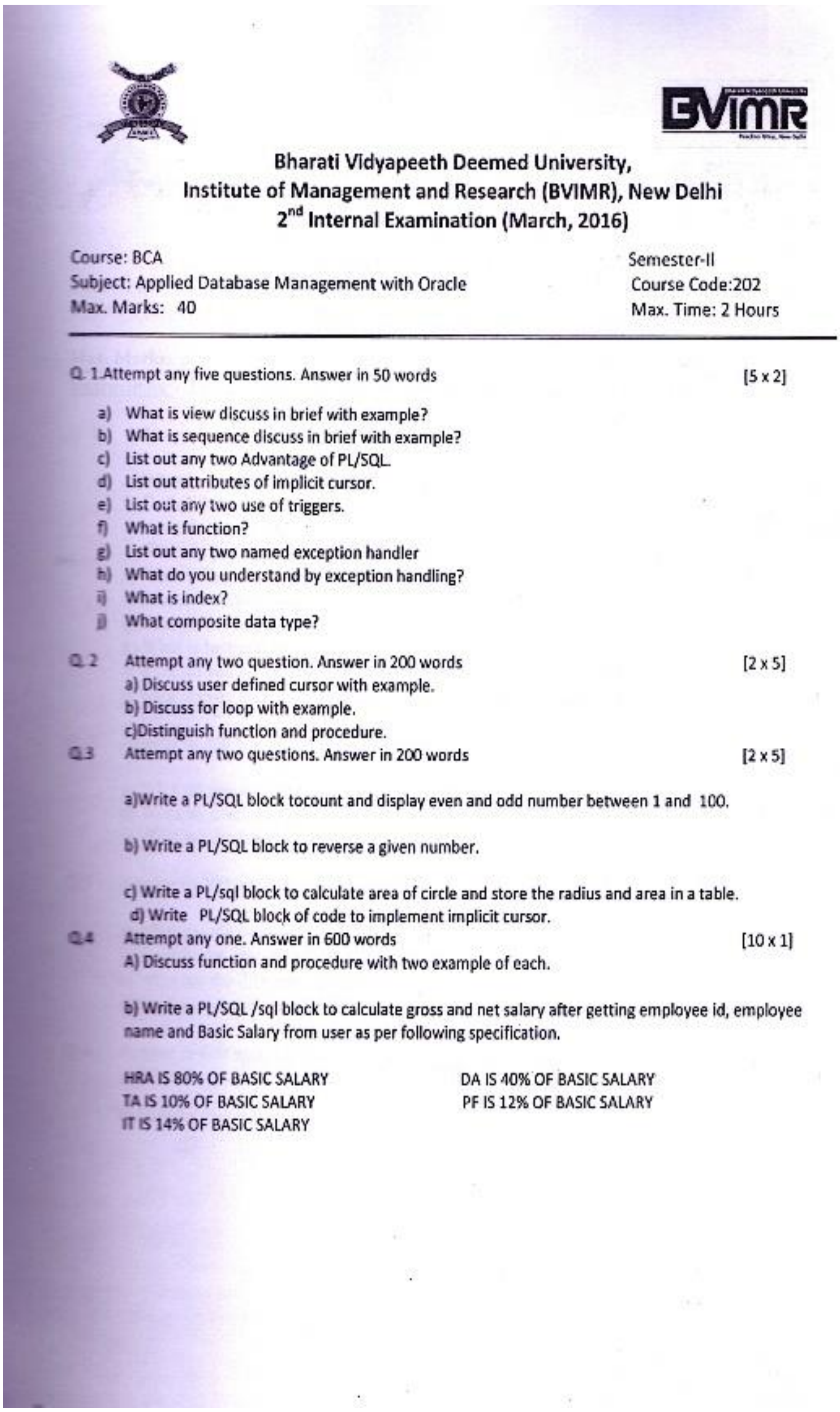
a) Explain all the option that can be used with select statement with example?

b) Explain the different advantages of using SQL and Pl/SQL

c) What are constraints in Oracle?

Q.4 Answer in 600 words **(Analytical Question / Case Study / Essay type question to test analytical and Comprehensive Skill)** [20 x 1]

Consider the following table:-
Branch (B_name,B_city)
Borrow (Loan_no,C_name,B_name,amount)
a) Create above table with proper constraint
b) Give the name of borrower having loan number 246
c) Give the name of borrower who has taken loan in the range of 5000 to 20000
d) Give the name of borrower whose name starts with letter "P"
e) List the total loan taken from M.G road branch
f) Display the borrower table in ascending order on column amount
g) Delete record of a customer Rajesh from borrow table



Bharati Vidyapeeth Deemed University,
Institute of Management and Research (BVIMR), New Delhi
2nd Internal Examination (March, 2016)

Course: BCA | Semester-II
Subject: Applied Database Management with Oracle | Course Code:202
Max. Marks: 40 | Max. Time: 2 Hours

Q. 1 Attempt any five questions. Answer in 50 words [5 x 2]

a) What is view discuss in brief with example?
b) What is sequence discuss in brief with example?
c) List out any two Advantage of PL/SQL.
d) List out attributes of implicit cursor.
e) List out any two use of triggers.
f) What is function?
g) List out any two named exception handler
h) What do you understand by exception handling?
i) What is index?
j) What composite data type?

Q. 2 Attempt any two question. Answer in 200 words [2 x 5]
a) Discuss user defined cursor with example.
b) Discuss for loop with example.
c)Distinguish function and procedure.

Q.3 Attempt any two questions. Answer in 200 words [2 x 5]

a)Write a PL/SQL block tocount and display even and odd number between 1 and 100.

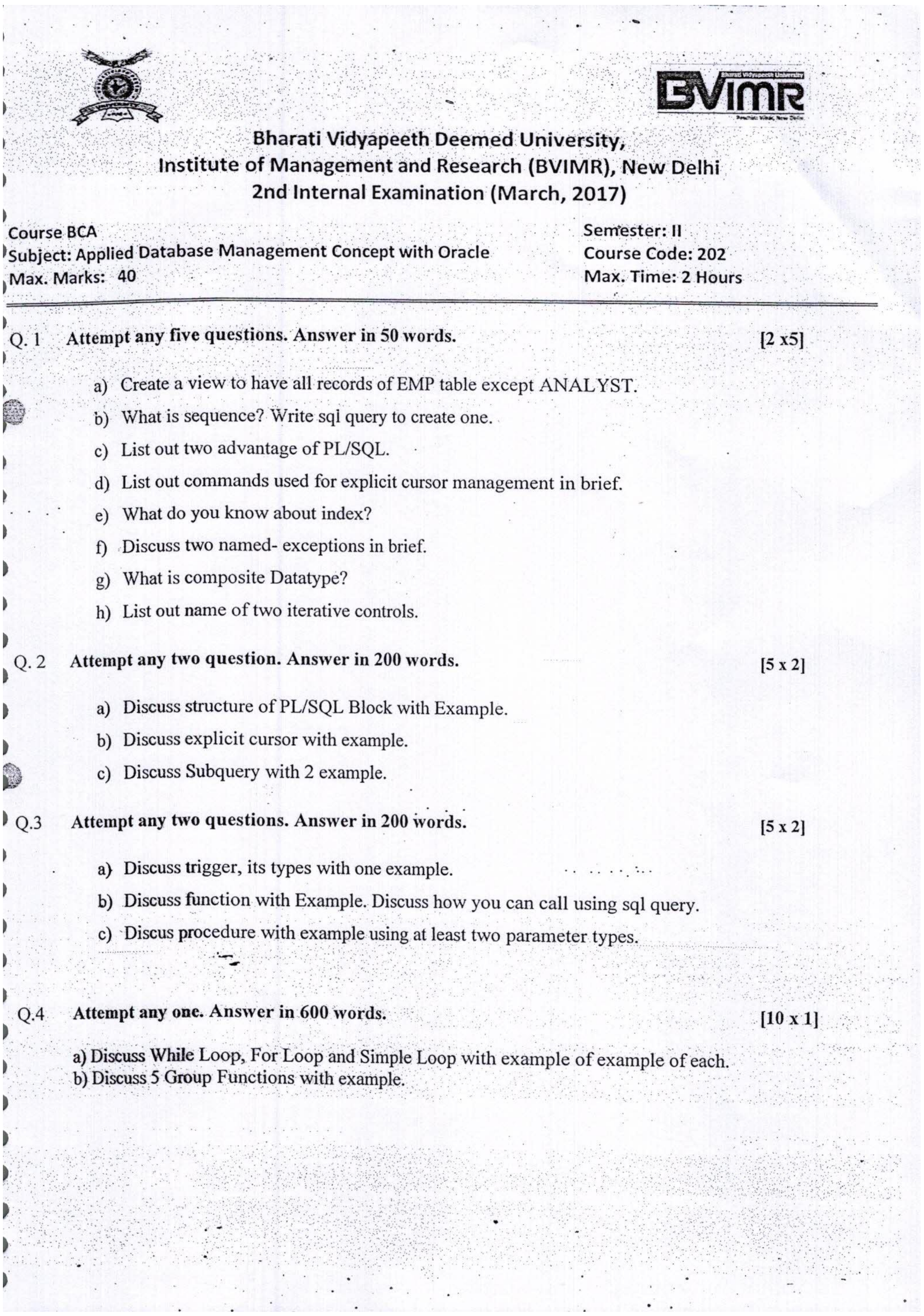
b) Write a PL/SQL block to reverse a given number.

c) Write a PL/sql block to calculate area of circle and store the radius and area in a table.
d) Write PL/SQL block of code to implement implicit cursor.

Q.4 Attempt any one. Answer in 600 words [10 x 1]
A) Discuss function and procedure with two example of each.

b) Write a PL/SQL /sql block to calculate gross and net salary after getting employee id, employee name and Basic Salary from user as per following specification.

HRA IS 80% OF BASIC SALARY	DA IS 40% OF BASIC SALARY
TA IS 10% OF BASIC SALARY	PF IS 12% OF BASIC SALARY
IT IS 14% OF BASIC SALARY	



Bharati Vidyapeeth Deemed University,
Institute of Management and Research (BVIMR), New Delhi
2nd Internal Examination (March, 2017)

Course BCA	Semester: II
Subject: Applied Database Management Concept with Oracle	Course Code: 202
Max. Marks: 40	Max. Time: 2 Hours

Q. 1 Attempt any five questions. Answer in 50 words. **[2 x5]**

a) Create a view to have all records of EMP table except ANALYST.
b) What is sequence? Write sql query to create one.
c) List out two advantage of PL/SQL.
d) List out commands used for explicit cursor management in brief.
e) What do you know about index?
f) Discuss two named- exceptions in brief.
g) What is composite Datatype?
h) List out name of two iterative controls.

Q. 2 Attempt any two question. Answer in 200 words. **[5 x 2]**

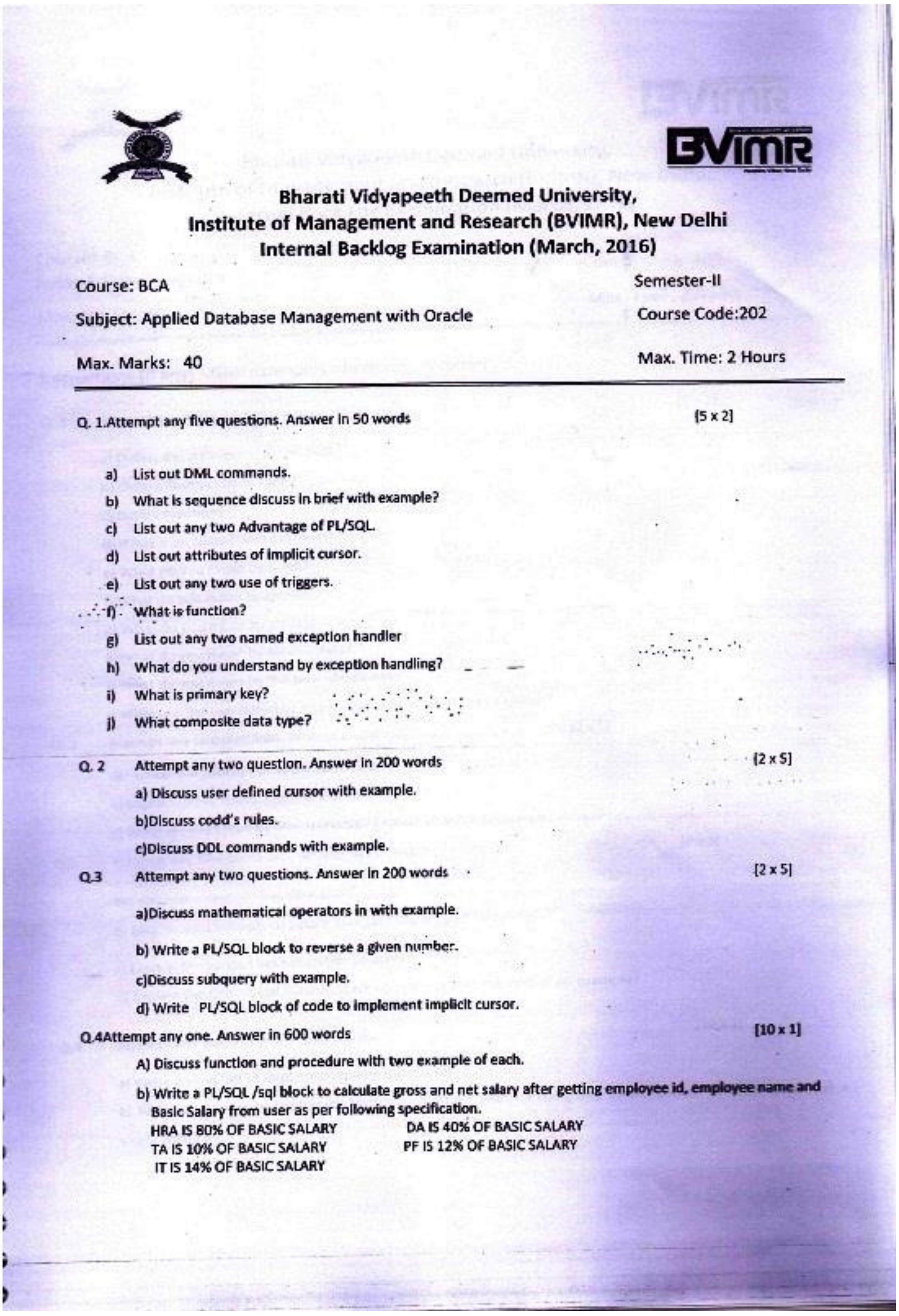
a) Discuss structure of PL/SQL Block with Example.
b) Discuss explicit cursor with example.
c) Discuss Subquery with 2 example.

Q.3 Attempt any two questions. Answer in 200 words. **[5 x 2]**

a) Discuss trigger, its types with one example.
b) Discuss function with Example. Discuss how you can call using sql query.
c) Discus procedure with example using at least two parameter types.

Q.4 Attempt any one. Answer in 600 words. **[10 x 1]**

a) Discuss While Loop, For Loop and Simple Loop with example of example of each.
b) Discuss 5 Group Functions with example.



Bharati Vidyapeeth Deemed University,
Institute of Management and Research (BVIMR), New Delhi
Internal Backlog Examination (March, 2016)

Course: BCA — Semester-II

Subject: Applied Database Management with Oracle — Course Code:202

Max. Marks: 40 — Max. Time: 2 Hours

Q. 1.Attempt any five questions. Answer in 50 words [5 x 2]

a) List out DML commands.
b) What is sequence discuss in brief with example?
c) List out any two Advantage of PL/SQL.
d) List out attributes of implicit cursor.
e) List out any two use of triggers.
f) What is function?
g) List out any two named exception handler
h) What do you understand by exception handling?
i) What is primary key?
j) What composite data type?

Q. 2 Attempt any two question. Answer in 200 words [2 x 5]

a) Discuss user defined cursor with example.
b)Discuss codd's rules.
c)Discuss DDL commands with example.

Q.3 Attempt any two questions. Answer in 200 words [2 x 5]

a)Discuss mathematical operators in with example.
b) Write a PL/SQL block to reverse a given number.
c)Discuss subquery with example.
d) Write PL/SQL block of code to implement implicit cursor.

Q.4Attempt any one. Answer in 600 words [10 x 1]

A) Discuss function and procedure with two example of each.
b) Write a PL/SQL /sql block to calculate gross and net salary after getting employee id, employee name and Basic Salary from user as per following specification.

HRA IS 80% OF BASIC SALARY	DA IS 40% OF BASIC SALARY
TA IS 10% OF BASIC SALARY	PF IS 12% OF BASIC SALARY
IT IS 14% OF BASIC SALARY	

University Question Papers

AMOOR- II (2014 Course) CBCS : SUMMER - 2016

Subject : Applied Database Management concepts using Oracle

Day : Wednesday
Date : 13/04/2016

29228

Time : 10.00 AM TO 01.00 PM
Max Marks : 100 Total Pages : 1

N. B. :

1) Attempt **ANY FOUR** questions from Section – **I**.
2) Attempt **ANY TWO** questions from Section – **II**.
3) Answers to both the sections should be written in the **SAME** answer book.

SECTION - I

Q. 1	Explain CODD's rules that qualify database as a relational database.	(15)
Q. 2	What is SQL? Explain the various data types and components of SQL.	(15)
Q. 3	Explain the Arithmetic, Relational and Logical operators in SQL.	(15)
Q. 4	Explain any 5 aggregate functions and any 5 string functions with example.	(15)
Q. 5	What is sequence? What is its use? Explain the syntax of creating sequence. Write a sequence to generate numbers from 1 to 50.	(15)
Q. 6	What is PL/SQL? Explain advantages of PL/SQL over SQL.	(15)
Q.7	What are stored procedures and functions? Explain the difference between them.	(15)

SECTION - II

Q. 8 Consider the following table structure:
Client_Master(Cno (PK), Name, City, Balance_Due)
Sales_Order(Order_No (PK), Order_Date, Cno (FK), Delivery_Date, Order_Status)
Write SQL Queries to:

a) Create above tables with proper constraints. (04)
b) Insert 2 records in both the tables. (02)
c) Find the names of clients having alphabet 'a' as a second letter in their name. (02)
d) Find the names of clients who stay in 'Mumbai' or 'Delhi'. (02)
e) Print the orders placed in the month of January. (02)
f) Display orders placed by client 'Ram'. (02)
g) Add column Phone Number to Client_Master table. (02)
h) Delete column order_status from Sales_Order table. (02)
i) Display the list of clients with their Balance_due in descending order. (02)

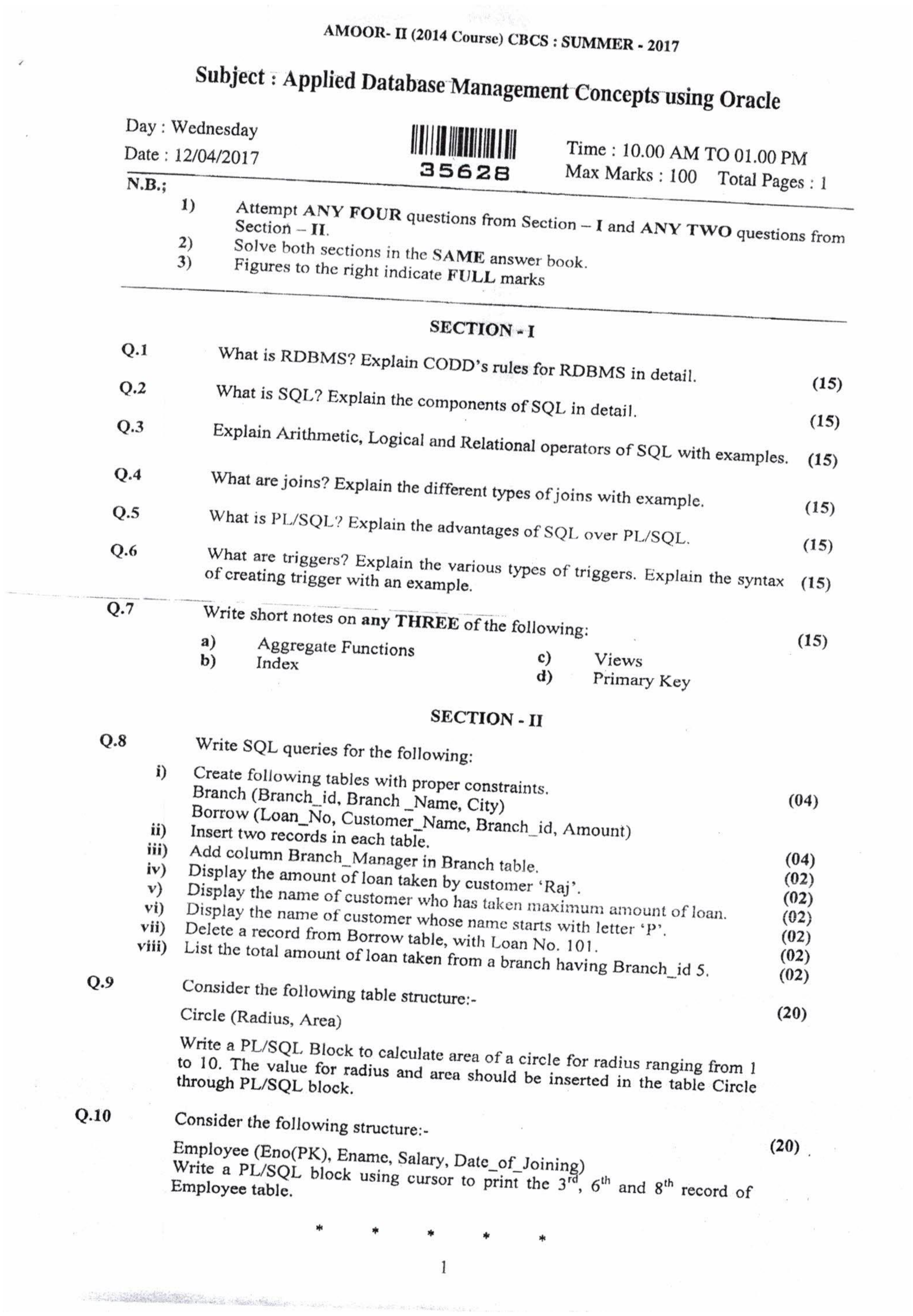
Q. 9 Write a PL/SQL Block to :

a) Write a PL/SQL Block to display numbers 1 to 10 using For Loop. (10)
b) Write a PL/SQL Block to generate Multiplication table of Number 5. (10)

Q.10 Write short notes on (**ANY TWO**) (20)

a) Views b) Set operators c) Cursor

* * * * *



AMOOR- II (2014 Course) CBCS : SUMMER - 2017

Subject : Applied Database Management Concepts using Oracle

Day : Wednesday
Date : 12/04/2017

35628

Time : 10.00 AM TO 01.00 PM
Max Marks : 100 Total Pages : 1

N.B.;

1) Attempt **ANY FOUR** questions from Section – **I** and **ANY TWO** questions from Section – **II**.
2) Solve both sections in the **SAME** answer book.
3) Figures to the right indicate **FULL** marks

SECTION - I

Q.1 What is RDBMS? Explain CODD's rules for RDBMS in detail. **(15)**

Q.2 What is SQL? Explain the components of SQL in detail. **(15)**

Q.3 Explain Arithmetic, Logical and Relational operators of SQL with examples. **(15)**

Q.4 What are joins? Explain the different types of joins with example. **(15)**

Q.5 What is PL/SQL? Explain the advantages of SQL over PL/SQL. **(15)**

Q.6 What are triggers? Explain the various types of triggers. Explain the syntax of creating trigger with an example. **(15)**

Q.7 Write short notes on **any THREE** of the following: **(15)**

a) Aggregate Functions
b) Index
c) Views
d) Primary Key

SECTION - II

Q.8 Write SQL queries for the following:

i) Create following tables with proper constraints. **(04)**
Branch (Branch_id, Branch _Name, City)
Borrow (Loan_No, Customer_Name, Branch_id, Amount)
ii) Insert two records in each table. **(04)**
iii) Add column Branch_Manager in Branch table. **(02)**
iv) Display the amount of loan taken by customer 'Raj'. **(02)**
v) Display the name of customer who has taken maximum amount of loan. **(02)**
vi) Display the name of customer whose name starts with letter 'P'. **(02)**
vii) Delete a record from Borrow table, with Loan No. 101. **(02)**
viii) List the total amount of loan taken from a branch having Branch_id 5. **(02)**

Q.9 Consider the following table structure:- **(20)**

Circle (Radius, Area)

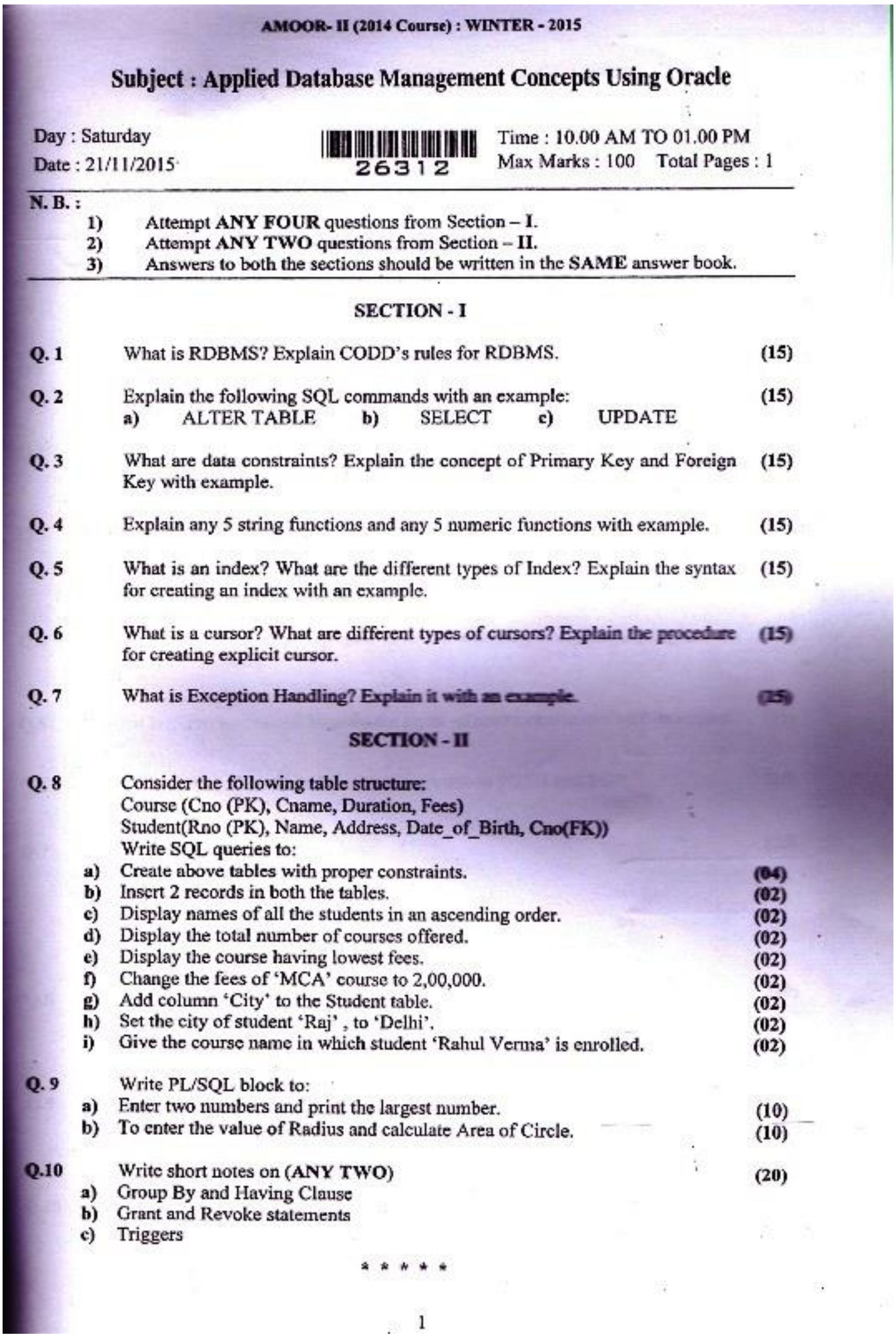
Write a PL/SQL Block to calculate area of a circle for radius ranging from 1 to 10. The value for radius and area should be inserted in the table Circle through PL/SQL block.

Q.10 Consider the following structure:- **(20)**

Employee (Eno(PK), Ename, Salary, Date_of_Joining)
Write a PL/SQL block using cursor to print the 3rd, 6th and 8th record of Employee table.

* * * * *

1



AMOOR- II (2014 Course) : WINTER - 2015

Subject : Applied Database Management Concepts Using Oracle

Day : Saturday
Date : 21/11/2015

26312

Time : 10.00 AM TO 01.00 PM
Max Marks : 100 Total Pages : 1

N. B. :

1) Attempt **ANY FOUR** questions from Section – I.
2) Attempt **ANY TWO** questions from Section – II.
3) Answers to both the sections should be written in the **SAME** answer book.

SECTION - I

Q. 1 What is RDBMS? Explain CODD's rules for RDBMS. **(15)**

Q. 2 Explain the following SQL commands with an example: **(15)**
a) ALTER TABLE **b)** SELECT **c)** UPDATE

Q. 3 What are data constraints? Explain the concept of Primary Key and Foreign Key with example. **(15)**

Q. 4 Explain any 5 string functions and any 5 numeric functions with example. **(15)**

Q. 5 What is an index? What are the different types of Index? Explain the syntax for creating an index with an example. **(15)**

Q. 6 What is a cursor? What are different types of cursors? Explain the procedure for creating explicit cursor. **(15)**

Q. 7 What is Exception Handling? Explain it with an example. **(15)**

SECTION - II

Q. 8 Consider the following table structure:
Course (Cno (PK), Cname, Duration, Fees)
Student(Rno (PK), Name, Address, Date_of_Birth, Cno(FK))
Write SQL queries to:

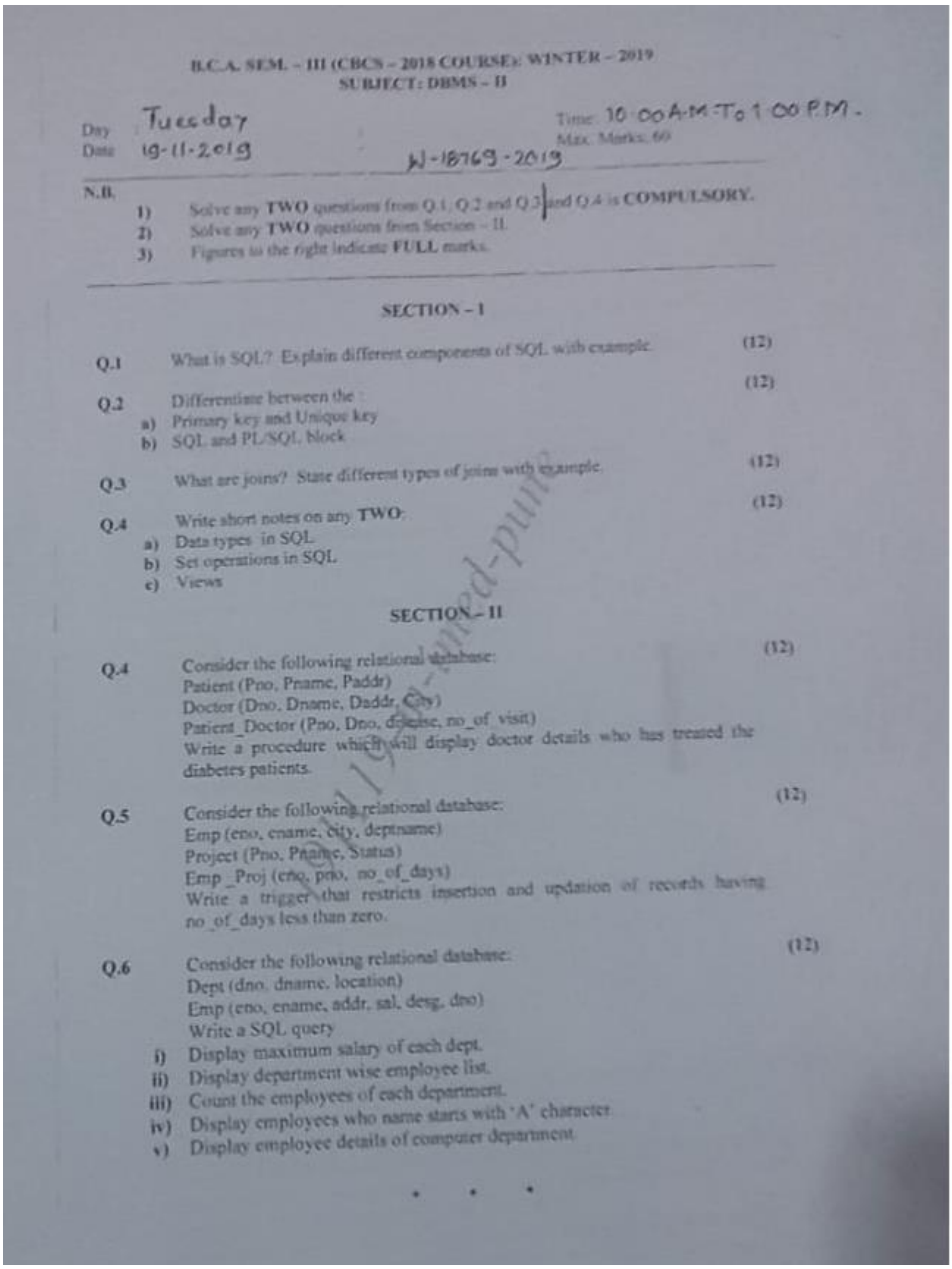
a) Create above tables with proper constraints. **(04)**
b) Insert 2 records in both the tables. **(02)**
c) Display names of all the students in an ascending order. **(02)**
d) Display the total number of courses offered. **(02)**
e) Display the course having lowest fees. **(02)**
f) Change the fees of 'MCA' course to 2,00,000. **(02)**
g) Add column 'City' to the Student table. **(02)**
h) Set the city of student 'Raj' , to 'Delhi'. **(02)**
i) Give the course name in which student 'Rahul Verma' is enrolled. **(02)**

Q. 9 Write PL/SQL block to:
a) Enter two numbers and print the largest number. **(10)**
b) To enter the value of Radius and calculate Area of Circle. **(10)**

Q.10 Write short notes on (**ANY TWO**) **(20)**
a) Group By and Having Clause
b) Grant and Revoke statements
c) Triggers

* * * * *

1



B.C.A. SEM. – III (CBCS – 2018 COURSE): WINTER – 2019
SUBJECT: DBMS – II

Day : Tuesday Time: 10·00 A·M· To 1·00 P.M.
Date : 19-11-2019 Max. Marks: 60

W-18769-2019

N.B.
1) Solve any **TWO** questions from Q.1, Q.2 and Q.3 and Q.4 is **COMPULSORY.**
2) Solve any **TWO** questions from Section – II.
3) Figures to the right indicate **FULL** marks.

SECTION – I

Q.1 What is SQL? Explain different components of SQL with example. (12)

Q.2 Differentiate between the : (12)
a) Primary key and Unique key
b) SQL and PL/SQL block

Q.3 What are joins? State different types of joins with example. (12)

Q.4 Write short notes on any **TWO**: (12)
a) Data types in SQL
b) Set operations in SQL
c) Views

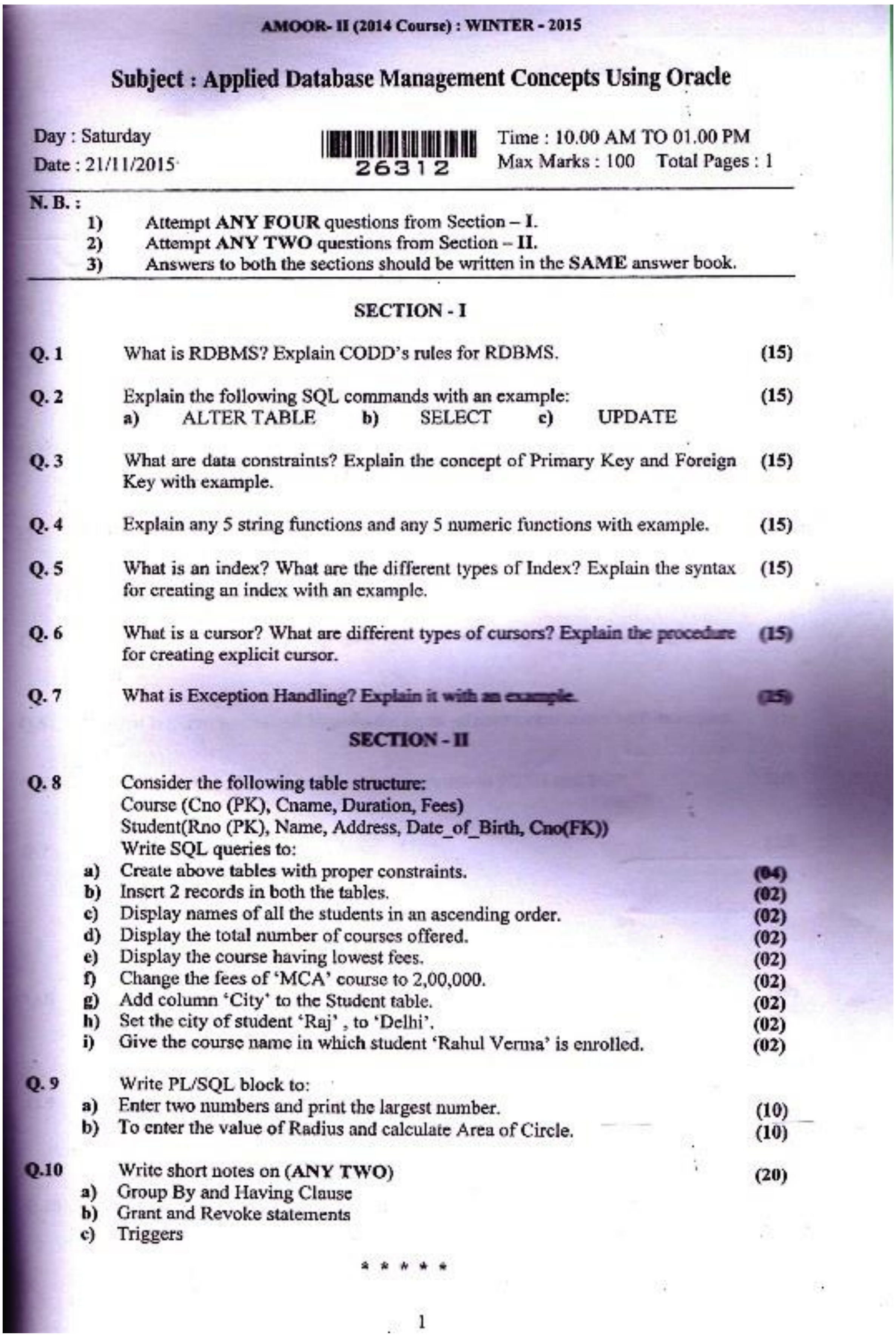
SECTION – II

Q.4 Consider the following relational database: (12)
Patient (Pno, Pname, Paddr)
Doctor (Dno, Dname, Daddr, City)
Patient_Doctor (Pno, Dno, disease, no_of_visit)
Write a procedure which will display doctor details who has treated the diabetes patients.

Q.5 Consider the following relational database: (12)
Emp (eno, ename, city, deptname)
Project (Pno, Pname, Status)
Emp_Proj (eno, pno, no_of_days)
Write a trigger that restricts insertion and updation of records having no_of_days less than zero.

Q.6 Consider the following relational database: (12)
Dept (dno, dname, location)
Emp (eno, ename, addr, sal, desg, dno)
Write a SQL query
i) Display maximum salary of each dept.
ii) Display department wise employee list.
iii) Count the employees of each department.
iv) Display employees who name starts with 'A' character
v) Display employee details of computer department.

* * *



AMOOR- II (2014 Course) : WINTER - 2015

Subject : Applied Database Management Concepts Using Oracle

Day : Saturday
Date : 21/11/2015

26312

Time : 10.00 AM TO 01.00 PM
Max Marks : 100 Total Pages : 1

N. B. :
1) Attempt **ANY FOUR** questions from Section – **I.**
2) Attempt **ANY TWO** questions from Section – **II.**
3) Answers to both the sections should be written in the **SAME** answer book.

SECTION - I

Q. 1 What is RDBMS? Explain CODD's rules for RDBMS. **(15)**

Q. 2 Explain the following SQL commands with an example: **(15)**
a) ALTER TABLE **b)** SELECT **c)** UPDATE

Q. 3 What are data constraints? Explain the concept of Primary Key and Foreign Key with example. **(15)**

Q. 4 Explain any 5 string functions and any 5 numeric functions with example. **(15)**

Q. 5 What is an index? What are the different types of Index? Explain the syntax for creating an index with an example. **(15)**

Q. 6 What is a cursor? What are different types of cursors? Explain the procedure for creating explicit cursor. **(15)**

Q. 7 What is Exception Handling? Explain it with an example. **(15)**

SECTION - II

Q. 8 Consider the following table structure:
Course (Cno (PK), Cname, Duration, Fees)
Student(Rno (PK), Name, Address, Date_of_Birth, Cno(FK))
Write SQL queries to:
a) Create above tables with proper constraints. **(04)**
b) Insert 2 records in both the tables. **(02)**
c) Display names of all the students in an ascending order. **(02)**
d) Display the total number of courses offered. **(02)**
e) Display the course having lowest fees. **(02)**
f) Change the fees of 'MCA' course to 2,00,000. **(02)**
g) Add column 'City' to the Student table. **(02)**
h) Set the city of student 'Raj' , to 'Delhi'. **(02)**
i) Give the course name in which student 'Rahul Verma' is enrolled. **(02)**

Q. 9 Write PL/SQL block to:
a) Enter two numbers and print the largest number. **(10)**
b) To enter the value of Radius and calculate Area of Circle. **(10)**

Q.10 Write short notes on (**ANY TWO**) **(20)**
a) Group By and Having Clause
b) Grant and Revoke statements
c) Triggers

* * * * *

DBMS-II

Lab Assignment-1

Name of the student
Class
Sem
ERPID
Date of submission
Mark Obtained (OUT OF 10)
Signature of Faculty

Note:-

1. **Write only one side of the paper**
2. **Use A4 sheet.**
3. **Tag all sheet with a ta**

Lab Assignments

Assignment No-1

1. List all information about all employees.
2. Display the name of all employees with their salary.
3. List all employees name who is working with department no 20.
4. List the name of all Salesman and Analyst.
5. Display the details of those employees who have joined before the end of September 1981.
6. List the employee's name, employee id who are Manager.
7. List the name and job of all employees who are not clerk.
8. List the name of employees whose employees no is 7369 ,752,7839, 7934 or 7788
9. List the employees details who does not belongs to department no 10 and 30
10. List the employees name and salary whose salary varies from 1000 to 2000.
11. List employees name who have joined before 30 June-1981 and after 30 Dec-1981
12. List the Commission and name of employees who are availing the commission.
13. List the name and designation of employee who does report any body.
14. List the details of employees whose salary is greater than 2000 and commission is Null.
15. List the employees details whose name start with 'S'.
16. List the employees name having 'I' as second character along with their Job.
17. List the employees name and date of joining in the descending order of date of joining the column title should be 'Date of Joining'.
18. List the employee name, salary, job, and department number and display it in ascending order of department no and descending order of salary.
19. List the employee's name, salary, PF, HRA, DA and Gross salary. Order the result is in ascending order of gross salary HRA is 50% of Salary, DA is 30% of Salary and PF is 10%.
20. List the department no and total no of employees in each department.
21. List the different job names available in emp table.
22. List department no and total salary payable in each department.
23. List the average salary and number of employees working in department no 20.
24. List the job and number of employees in each job the result should be in descending order of number of employees.

25. List the total salary, maximum salary, minimum salary and average salary of employee’s job wise for department no 10 only.
26. List of the average salary of each job excluding manager.
27. List the average salary of each job within each department.
28. List the average salary of all departments in which more than 5 people working.
29. List the jobs of all employees where maximum salary is greater than equal to 5000.
30. List the total salary average salary of the employees job wise for department no 20 and display only those rows having average salary greater than 1000.

Assignment-2

1. Retrieve records from any two tables without using any condition.
2. Display the number of employees working in each department. Display the department information also even if no employee working to that department.
3. List the name of manager along with employee’s name, salary and manager code.
4. List the employee’s details who have joined the company before their manager.
5. Display different job available in the department no 20 and 30 using set operator.
6. Display all jobs commission in department no 20 and 30 by using set operator.
7. Select all jobs unique in department no 20 by using set operator.
8. List the employees name department no and department name from emp and dept table based on department number make sure that only those record select whose designation is manager.
9. Display the entire department no from emp and dept table without eliminating the duplicate data.
10. Fetch the employees no and employees name from emp table where department no is 10 and 30 and order the based-on employee name using set operator.

Assignment -3

1. Display the absolute value of -101
2. Display the smallest value of 16.0 and 59.2
3. Display the greatest integer value of 16.0 and 59.2
4. Calculate the remainder for given tow number (213, 9).
5. Calculate the power of two number entered by user at runtime of query
6. Enter a number and display whether it is negative or positive.
7. Calculate square root of a given number
8. Enter a float number and truncate it up to 1, 2 and -2 place of decimal.
9. Find the round of value of 563.456 up to 2, 0 and -2 place of decimal.
10. Accept two numbers and display it corresponding character with the appropriate title.
11. Input two names and concatenate it separated by two spaces.
12. Display all employees name with first character in upper case from emp table.
13. Display department name in right Alignment format
14. Display department name in upper and lower cases.
15. Extract character S and A from the left and R and N from right of the employees name from EMP table.
16. Suffix all salary with # sign.
17. Change all the occurrence of CL with P in Job domain.
18. Display all information of employee whose name has the second character as A.
19. Display all ASCII code of the character entered by the user.
20. Display the all employees names along with the location of the character A in the Employees name from emp table where job is Clerk
21. Find the employee name with maximum number of character in it .
22. Display the salary and increment of those employees who are getting salary in the three figures salary has to be incremented by 50% of original salary.
23. Display the details of employee SCOTT
24. Display the name ,salary , commission and gross pay for all the employees
25. Display first three character of employee name and their HRA (sal * 123) truncated to 2 decimal places.
26. List all employee names, who have more than 20 years of experience in the company.
27. Display the name and job for every employee while displaying hob ‘CLERK’ should be display as Lower Divisional CLERK (LDC) the other job title should be as they are.

28. Display the greatest and lowest value among salary and commission for all employees.

Solution of Test Paper

Ans1. RDBMS- RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.

Codd's Rule for Relational DBMS

E.F Codd was a Computer Scientist who invented the **Relational model** for Database management. Based on relational model, the **Relational database** was created. Codd proposed 13 rules popularly known as **Codd's 12 rules** to test DBMS's concept against his relational model. Codd's rule actualy define what quality a DBMS requires in order to become a Relational Database Management System(RDBMS). Till now, there is hardly any commercial product that follows all the 13 Codd's rules. Even **Oracle** follows only eight and half(8.5) out of 13. The Codd's 12 rules are as follows.

Rule zero

This rule states that for a system to qualify as an **RDBMS**, it must be able to manage database entirely through the relational capabilities.

Rule 1: Information rule

All information(including metadata) is to be represented as stored data in cells of tables. The rows and columns have to be strictly unordered.

Rule 2: Guaranted Access

Each unique piece of data(atomic value) should be accesible by : **Table Name + Primary Key(Row) + Attribute(column)**.

NOTE: Ability to directly access via POINTER is a violation of this rule.

Rule 3: Systematic treatment of NULL

Null has several meanings, it can mean missing data, not applicable or no value. It should be handled consistently. Also, Primary key must not be null, ever. Expression on NULL must give null.

Rule 4: Active Online Catalog

Database dictionary(catalog) is the structure description of the complete **Database** and it must be stored online. The Catalog must be governed by same rules as rest of the database. The same query language should be used on catalog as used to query database.

Rule 5: Powerful and Well-Structured Language

One well structured language must be there to provide all manners of access to the data stored in the database. Example: **SQL**, etc. If the database allows access to the data without the use of this language, then that is a violation.

Rule 6: View Updation Rule

All the view that are theoretically updatable should be updatable by the system as well.

Rule 7: Relational Level Operation

There must be Insert, Delete, Update operations at each level of relations. Set operation like Union, Intersection and minus should also be supported.

Rule 8: Physical Data Independence

The physical storage of data should not matter to the system. If say, some file supporting table is renamed or moved from one disk to another, it should not effect the application.

Rule 9: Logical Data Independence

If there is change in the logical structure(table structures) of the database the user view of data should not change. Say, if a table is split into two tables, a new view should give result as the join of the two tables. This rule is most difficult to satisfy.

Rule 10: Integrity Independence

The database should be able to enforce its own integrity rather than using other programs. Key and Check constraints, trigger etc, should be stored in Data Dictionary. This also make **RDBMS**independent of front-end.

Rule 11: Distribution Independence

A database should work properly regardless of its distribution across a network. Even if a database is geographically distributed, with data stored in pieces, the end user should get an impression that it is stored at the same place. This lays the foundation of **distributed database**.

Rule 12: Nonsubversion Rule

If low level access is allowed to a system it should not be able to subvert or bypass integrity rules to change the data. This can be achieved by some sort of looking or encryption.

Ans 2. Alter table-ALTER command alter command is used for altering the table structure, such as,

- to add a column to existing table
- to rename any existing column
- to change datatype of any column or to modify its size.
- to drop a column from the table.

ALTER Command: Add a new Column

Using ALTER command we can add a column to any existing table. Following is the syntax,

```
ALTER TABLE table_name ADD(    column_name datatype);
```

Here is an Example for this,

```
ALTER TABLE student ADD(
    address VARCHAR(200)
);
```

The above command will add a new column address to the table student, which will hold data of type varchar which is nothing but string, of length 200.

ALTER Command: Add multiple new Columns

Using ALTER command we can even add multiple new columns to any existing table. Following is the syntax,

```
ALTER TABLE table_name ADD(
    column_name1 datatype1,
    column-name2 datatype2,
    column-name3 datatype3);
```

Here is an Example for this,

```
ALTER TABLE student ADD(
    father_name VARCHAR(60),
    mother_name VARCHAR(60),
    dob DATE);
```

The above command will add three new columns to the **student** table

ALTER Command: Add Column with default value

ALTER command can add a new column to an existing table with a default value too. The default value is used when no value is inserted in the column. Following is the syntax,

```
ALTER TABLE table_name ADD(
    column-name1 datatype1 DEFAULT some_value
);
```

Here is an Example for this,

```
ALTER TABLE student ADD(
    dob DATE DEFAULT '01-Jan-99'
);
```

The above command will add a new column with a preset default value to the table **student**.

ALTER Command: Modify an existing Column

ALTER command can also be used to modify data type of any existing column. Following is the syntax,

```
ALTER TABLE table_name modify(
    column_name datatype
);
```

Here is an Example for this,

```
ALTER TABLE student MODIFY(
    address varchar(300));
```

Remember we added a new column `address` in the beginning? The above command will modify the `address` column of the **student** table, to now hold upto 300 characters.

ALTER Command: Rename a Column

Using ALTER command you can rename an existing column. Following is the syntax,

ALTER TABLE table_name RENAME   old_column_name TO new_column_name;

Here is an example for this,

ALTER TABLE student RENAME   address TO location;

The above command will rename `address` column to `location`.

ALTER Command: Drop a Column

ALTER command can also be used to drop or remove columns. Following is the syntax,

ALTER TABLE table_name DROP(column_name);

Here is an example for this,

ALTER TABLE student DROP(address);

The above command will drop the `address` column from the table **student**.

Select -SELECT SQL Query

SELECT query is used to retrieve data from a table. It is the most used SQL query. We can retrieve complete table data, or partial by specifying conditions using the WHERE clause.

Syntax of SELECT query

SELECT query is used to retieve records from a table. We can specify the names of the columns which we want in the resultset.

```
SELECT
    column_name1,
    column_name2,
    column_name3,
    ...
    column_nameN
    FROM table_name;
```

SELECT s_id, name, age FROM student;

The above query will fetch information of `s_id`, `name` and `age` columns of the **student** table and display them,

s_id	**name**	**age**
101	Adam	15
102	Alex	18
103	Abhi	17
104	Ankit	22

- `SELECT` statement uses `*` character to retrieve all records from a table, for all the columns.

SELECT * FROM student;

The statement below will give all the details of the student whose name is Abhi.

SELECT * FROM student WHERE name = 'Abhi';

SELECT eid, name, salary+3000 FROM employee;

- The above command will display a new column in the result, with **3000** added into existing salaries of the employees.

Update- The UPDATE statement is used to modify the existing records in a table..

UPDATE *table_name*

SET *column1* = *value1*, *column2* = *value2*, ...

WHERE *condition*;

t is the WHERE clause that determines how many records that will be updated.

The following SQL statement will update the contactname to "Juan" for all records where country is "Mexico":

Example

```
UPDATE Customers SET ContactName='Juan' WHERE Country='Mexico';
```

Ans3. Data Constraints- Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

Following are some of the most commonly used constraints available in SQL. These constraints have already been discussed in SQL - RDBMS Conceptschapter, but it’s worth to revise them at this point.

- NOT NULL Constraint − Ensures that a column cannot have NULL value.
- DEFAULT Constraint − Provides a default value for a column when none is specified.
- UNIQUE Constraint − Ensures that all values in a column are different.
- PRIMARY Key − Uniquely identifies each row/record in a database table.
- FOREIGN Key − Uniquely identifies a row/record in any of the given database table.
- CHECK Constraint − The CHECK constraint ensures that all the values in a column satisfies certain conditions.

Integrity Constraints

Integrity constraints are used to ensure accuracy and consistency of the data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity.

There are many types of integrity constraints that play a role in Referential Integrity (RI). These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints which are mentioned above.

Defining Primary key and Foreign key

```
 --Create Parent Table
CREATE TABLE Department  ( DeptID int PRIMARY KEY, --define primary key
 Name varchar (50) NOT NULL,
 Address varchar(100) NULL
)
--Create Child Table
CREATE TABLE Employee
 (
 EmpID int PRIMARY KEY, --define primary key
 Name varchar (50) NOT NULL,
 Salary int NULL,
 --define foreign key
 DeptID int FOREIGN KEY REFERENCES Department(DeptID)
 )
```

Ans 4. SQL | Character Functions with Examples

Character functions accept character inputs and can return either characters or number values as output. SQL provides a number of different character datatypes which includes – CHAR, VARCHAR, VARCHAR2, LONG, RAW, and LONG RAW. The various datatypes are categorized into three different datatypes :

VARCHAR2 – A variable-length character datatype whose data is converted by the RDBMS.

CHAR – The fixed-length datatype.

RAW – A variable-length datatype whose data is not converted by the RDBMS, but left in "raw" form.

Note : When a character function returns a character value, that value is always of type VARCHAR2 (variable length), with the following two exceptions: UPPER and LOWER. These functions convert to upper and to lower case, respectively, and return the CHAR values (fixed length) if the strings they are called on to convert are **fixed-length** CHAR arguments.

Character Functions

SQL provides a rich set of character functions that allow you to get information about strings and modify the contents of those strings in multiple ways. Character functions are of the following two types:
1. Case-Manipulative Functions (LOWER, UPPER and INITCAP)
2. Character-Manipulative Functions (CONCAT, LENGTH, SUBSTR, INSTR, LPAD, RPAD, TRIM and REPLACE)

Case-Manipulative Functions

LOWER : This function converts alpha character values to lowercase. LOWER will actually return a fixed-length string if the incoming string is fixed-length. LOWER will not change any characters in the string that are not letters, since case is irrelevant for numbers and special characters, such as the dollar sign ($) or modulus (%).
Syntax:

LOWER(SQL course)

UPPER : This function converts alpha character values to uppercase. Also UPPER function too, will actually return a fixed-length string if the incoming string is fixed-length. UPPER will not change any characters in the string that are not letters, since case is irrelevant for numbers and special characters, such as the dollar sign ($) or modulus (%).
Syntax:

UPPER(SQL course)

CONCAT : This function always appends (concatenates) string2 to the end of string1. If either of the string is NULL, CONCAT function returns the non-NULL argument. If both strings are NULL, CONCAT returns NULL.
Syntax:

CONCAT('String1', 'String2')

LENGTH : This function returns the length of the input string. If the input string is NULL, then LENGTH function returns NULL and not Zero. Also, if the input string contains extra spaces at the start, or in between or at the end of the string, then the LENGTH function includes the extra spaces too and returns the complete length of the string.
Syntax:

LENGTH(Column|Expression)

SUBSTR : This function returns a portion of a string from a given start point to an end point. If a substring length is not given, then SUBSTR returns all the characters till the end of string (from the starting position specified).
Syntax:

SUBSTR('String',start-index,length_of_extracted_string)

Numeric

Numeric Functions are used to perform operations on numbers and return numbers. Following are the numeric functions defined in SQL:

ABS(): It returns the absolute value of a number.

Syntax: SELECT ABS(-243.5);

Output: 243.5

ACOS(): It returns the cosine of a number.

Syntax: SELECT ACOS(0.25);

Output: 1.318116071652818

ASIN(): It returns the arc sine of a number.

Syntax: SELECT ASIN(0.25);

Output: 0.25268025514207865

CEIL(): It returns the smallest integer value that is greater than or equal to a number.

Syntax: SELECT CEIL(25.75);

Output: 26

CEILING(): It returns the smallest integer value that is greater than or equal to a number.

Syntax: SELECT CEILING(25.75);

Output: 26

COS(): It returns the cosine of a number.

Syntax: SELECT COS(30);

Output: 0.15425144988758405

COT(): It returns the cotangent of a number.

Syntax: SELECT COT(6);

Output: -3.436353004180128

DEGREES(): It converts a radian value into degrees.

Syntax: SELECT DEGREES(1.5);

GREATEST(): It returns the greatest value in a list of expressions.

Syntax: SELECT GREATEST(30, 2, 36, 81, 125);

Output: 125

LEAST(): It returns the smallest value in a list of expressions.

Syntax: SELECT LEAST(30, 2, 36, 81, 125);

Ans 5.Index and types of Index

Indexing in Databases

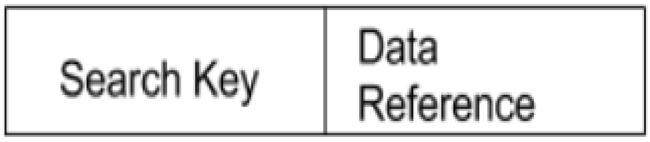
Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed.

An index or database index is a data structure which is used to quickly locate and access the data in a database table.

Indexes are created using some database columns.

The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly (Note that the data may or may not be stored in sorted order).

The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.



Structure of an index

There are two kinds of indices:

Ordered indices: Indices are based on a sorted ordering of the values.

Hash indices: Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by function called a hash function.

There is no comparison between both the techniques, it depends on the database application on which it is being applied.

- **Access Types**: e.g. value based search, range access, etc.

- **Access Time**: Time to find particular data element or set of elements.
- **Insertion Time**: Time taken to find the appropriate space and insert a new data.
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: Additional space required by the index.

Indexing Methods

Ordered Indices

The indices are usually sorted so that the searching is faster. The indices which are sorted are known as ordered indices.

If the search key of any index specifies same order as the sequential order of the file, it is known as primary index or clustering index.
Note: The search key of a primary index is usually the primary key, but it is not necessarily so.

If the search key of any index specifies an order different from the sequential order of the file, it is called the secondary index or non-clustering index.

Clustered Indexing

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rdsemester students etc are grouped.

Clustered index sorted according to first name (Search key)

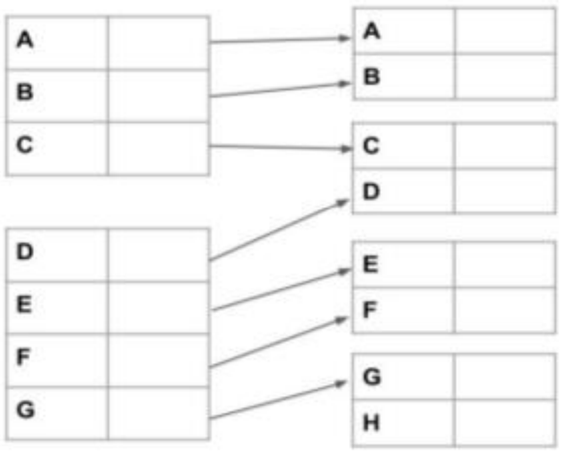
Primary Index

In this case, the data is sorted according to the search key. It induces sequential file organisation.
In this case, the primary key of the database table is used to create the index. As primary keys are unique and are stored in sorted manner, the performance of searching operation is quite efficient. The primary index is classified into two types : **Dense Index** and **Sparse Index**.

(I) Dense Index :

For every search key value in the data file, there is an index record.

This record contains the search key and also a reference to the first data record with that search key value.



Dense Index

Sparse Index :

The index record appears only for a few items in the data file. Each item points to a block as shown.

To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.

We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.

Ans 6. 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

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;
```

```
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;
```

Part-II

Ans 8

a) create table Course(cno integer primary key,Cname varchar2(20),Duration time, fees number(5,3))

Create table student (Rno integer primary key, name varchar2(20), address varchar2(20),date_of_birth date ,Cno integer references Course(cno))

b) insert into course values(12,’ROHIT’,’20:20:21’,5000)

insert into student values(122,’HARISH’,’DELHI’,’02-FEB-2008’,12)

c) Select name from student order by name;

d) Select count(Cno) as total_no from Course;

e) Select min(fees) from course;

f) Update course set fees=’2,00,000’ where Cname =’MCA’;

g) Alter table student add city varchar2(20);

h) Update student set city=’Delhi’ where name=’Raj’;

i) Select cname from course where cno=(select cno from student where name=’Rahul Verma’);

Ans 9 a) DECLARE   N NUMBER;

```
   M NUMBER;
BEGIN
```

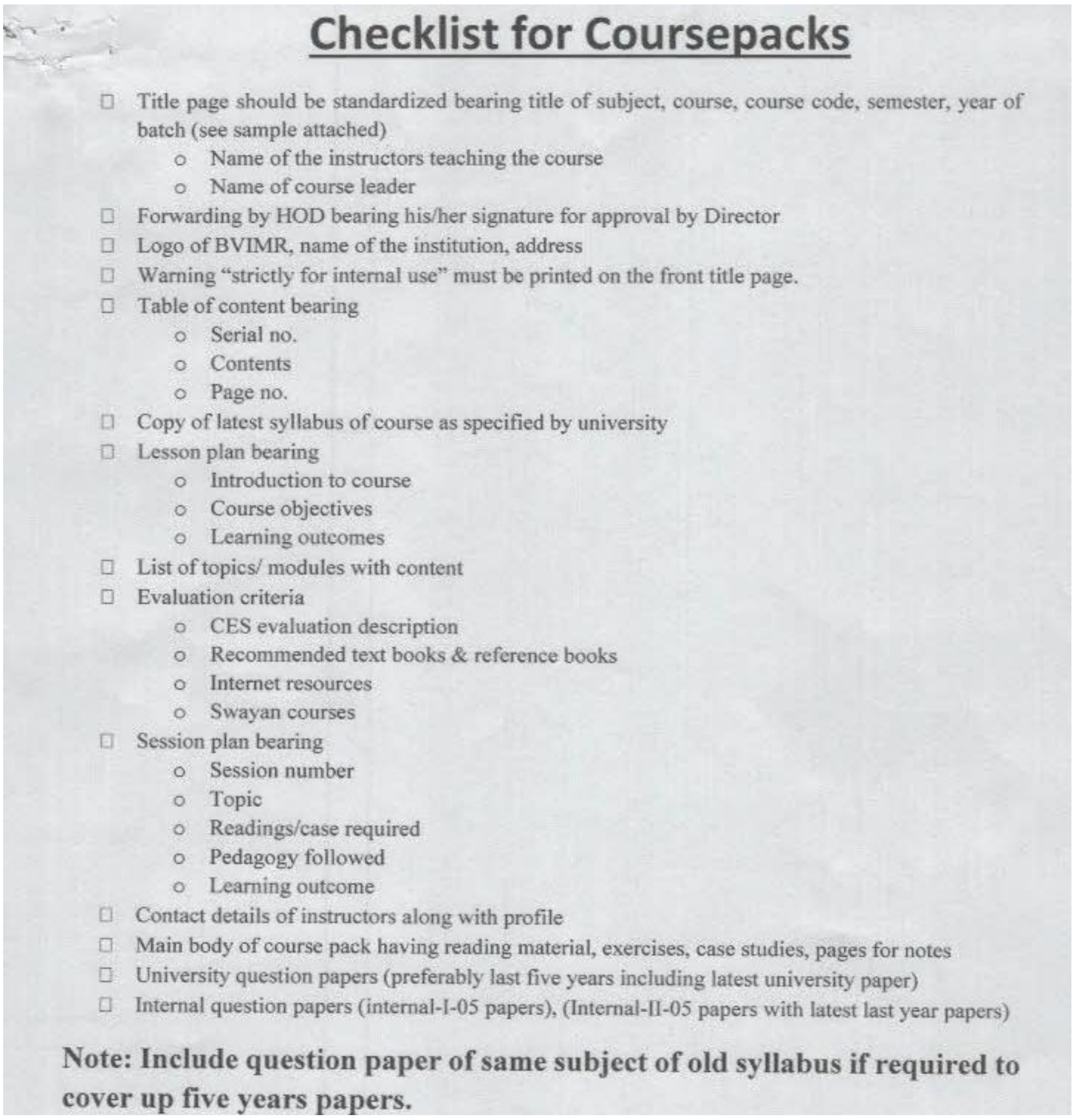
```
   DBMS_OUTPUT.PUT_LINE('ENTER A NUMBER');
   N:=&NUMBER;
   DBMS_OUTPUT.PUT_LINE('ENTER A NUMBER');
   M:=&NUMBER;
IF N<M THEN
   DBMS_OUTPUT.PUT_LINE(''|| N ||' IS GREATER THAN '|| M ||'');
ELSE
   DBMS_OUTPUT.PUT_LINE(''|| M ||' IS GREATER THAN '|| N ||'');
END IF;
END;
/
```

To enter the value of radius and calculate area of circle

```
DECLARE
     area NUMBER(6, 2) ;
             radius NUMBER(1) := 3;
             pi CONSTANT NUMBER(3, 2) := 3.14;
     BEGIN
             area := pi * radius * radius;
             perimeter := 2 * pi * radius;
             dbms_output.Put_line('Area = ' || area);
             dbms_output.Put_line(' Perimeter = ' || perimeter); END;
```

Mahesh Kumar Chaubey, B.Sc., MCA, MTech(CS), Pursuing PhD, HOD BCA, Technical Head ERP at BVIMR, , New Delhi, Oracle Certified Trainer, Having 12+ years of Teaching experience in field of computer application , Area of interest is Database Design and Data Mining.

Ajay Kumar working as assistant professor in Bharati Vidyapeeth (Deemed to be university) Institute of management & research New Delhi. He has 10 years' experience in teaching and his area of interest is Network Security and machine learning . He is Pursuing Ph.D from Mewar University..His subject areas are Computer network, database management system



Checklist for Coursepacks

- ☐ Title page should be standardized bearing title of subject, course, course code, semester, year of batch (see sample attached)
  - o Name of the instructors teaching the course
  - o Name of course leader
- ☐ Forwarding by HOD bearing his/her signature for approval by Director
- ☐ Logo of BVIMR, name of the institution, address
- ☐ Warning "strictly for internal use" must be printed on the front title page.
- ☐ Table of content bearing
  - o Serial no.
  - o Contents
  - o Page no.
- ☐ Copy of latest syllabus of course as specified by university
- ☐ Lesson plan bearing
  - o Introduction to course
  - o Course objectives
  - o Learning outcomes
- ☐ List of topics/ modules with content
- ☐ Evaluation criteria
  - o CES evaluation description
  - o Recommended text books & reference books
  - o Internet resources
  - o Swayan courses
- ☐ Session plan bearing
  - o Session number
  - o Topic
  - o Readings/case required
  - o Pedagogy followed
  - o Learning outcome
- ☐ Contact details of instructors along with profile
- ☐ Main body of course pack having reading material, exercises, case studies, pages for notes
- ☐ University question papers (preferably last five years including latest university paper)
- ☐ Internal question papers (internal-I-05 papers), (Internal-II-05 papers with latest last year papers)

Note: Include question paper of same subject of old syllabus if required to cover up five years papers.

Declaration by Faculty

I Mahesh Kumar Chaubey and Ajay Kumar, Designation asst. professor Teaching DBMS II subject in BCA course III Sem have incorporated all the necessary pages section/quotations papers mentioned in this check list above.

NOTES