SECTION V: PL / SQL

15. INTRODUCTION TO PL/SQL

Though SQL is the natural language of the DBA, it suffers from various inherent **disadvantages**, when used as a conventional programming language.

1. SQL does not have any **procedural capabilities** i.e. SQL does not provide the programming techniques of condition checking, looping and branching that is vital for data testing before its permanent storage
2. SQL statements are passed to the Oracle Engine **one** at a time. Each time an SQL statement is executed, a call is made to the engine's resources. This adds to the traffic on the network, thereby decreasing the speed of data processing, especially in a multi-user environment
3. While processing an SQL sentence if an error occurs, the Oracle engine displays its own error messages. SQL has no facility for programmed handling of errors that arise during the manipulation of data

Although SQL is a very powerful tool, its set of disadvantages prevent, it from being a fully structured programming language. For a fully structured programming language, Oracle provides **PL/SQL**.

As the name suggests, PL/SQL is a **superset** of SQL. PL/SQL is a block-structured language that enables developers to combine the power of SQL with procedural statements. PL/SQL bridges the gap between database technology and procedural programming languages.

ADVANTAGES OF PL/SQL

1. PL/SQL is development tool that not only supports SQL data manipulation but also provides facilities of conditional checking, branching and looping
2. PL/SQL sends an **entire block** of SQL statements to the Oracle engine all in one go. Communication between the program block and the Oracle engine reduces considerably, reducing network traffic. Since the Oracle engine got the SQL statements as a single block, it processes this code much faster than if it got the code one sentence at a time. There is a definite improvement in the performance time of the Oracle engine. As an entire block of SQL code is passed to the Oracle engine at one time for execution, all changes made to the data in the table are **done** or **undone**, in one go
3. PL/SQL also permits dealing with errors as required, and facilitates displaying user-friendly messages, when errors are encountered
4. PL/SQL allows declaration and use of variables in blocks of code. These variables can be used to store intermediate results of a query for later processing, or calculate values and insert them into an Oracle table later. PL/SQL variables can be used anywhere, either in SQL statements or in PL/SQL blocks
5. Via PL/SQL, all sorts of calculations can be done quickly and efficiently without the use of the Oracle engine. This considerably improves transaction performance
6. Applications written in PL/SQL are portable to any computer hardware and operating system, where Oracle is operational. Hence, PL/SQL code blocks written for a DOS version of Oracle will run on its Linux / UNIX version, **without** any modifications at all

THE GENERIC PL/SQL BLOCK

Every programming environment allows the creation of structured, logical blocks of code that describe processes, which have to be applied to data. Once these blocks are passed to the environment, the processes described are applied to data, suitable data manipulation takes place, and useful output is obtained.

PL/SQL permits the creation of structured logical blocks of code that describe processes, which have to be applied to data. A single PL/SQL code block consists of a set of SQL statements, clubbed together, and passed to the Oracle engine entirely. This block has to be logically grouped together for the engine to recognize it as a singular code block. A PL/SQL block has a definite structure, which can be divided into sections. The sections of a PL/SQL block are:

- The Declare section,
- The Master Begin and End section that **also** (optionally) **contains** an Exception section.

Each of these is explained below:

The Declare Section

Code blocks start with a declaration section, in which, memory variables and other Oracle objects can be declared, and if required initialized. Once declared, they can be used in SQL statements for data manipulation.

The Begin Section

It consists of a set of SQL **and** PL/SQL statements, which describe processes that have to be applied to table data. Actual data manipulation, retrieval, looping and branching constructs are specified in this section.

The Exception Section

This section deals with handling of errors that arise during execution of the data manipulation statements, which make up the PL/SQL code block. Errors can arise due to syntax, logic and/or validation rule violation.

The End Section

This marks the end of a PL/SQL block.

A PL/SQL code block can be diagrammatically represented as follows:

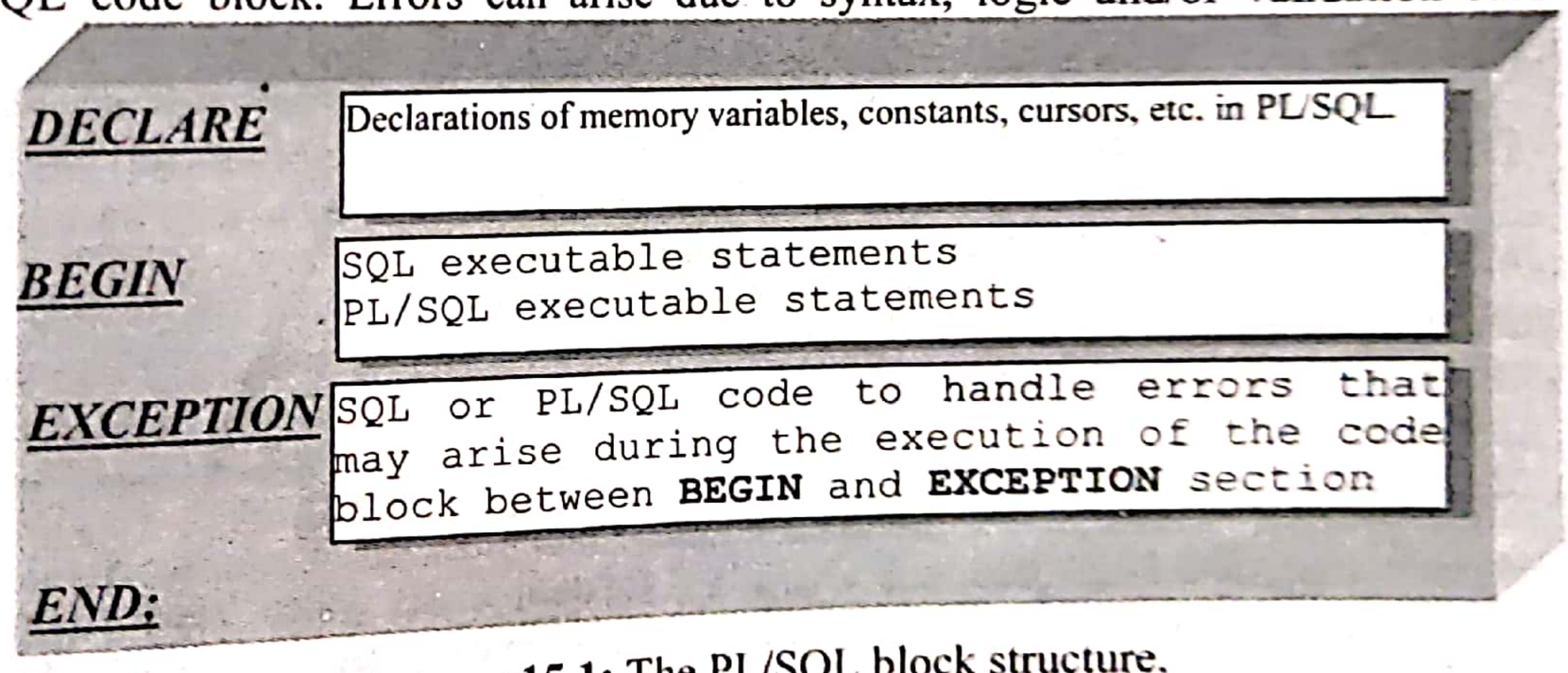

Diagram 15.1: The PL/SQL block structure.

THE PL/SQL EXECUTION ENVIRONMENT

Wherever PL/SQL technology is required (**i.e.** in the RDBMS core or in its tools), the PL/SQL engine accepts any valid PL/SQL block as input.

PL/SQL In The Oracle Engine

The PL/SQL engine resides in the Oracle engine, the Oracle engine can process not only single SQL statements but also entire PL/SQL blocks.

These blocks are sent to the PL/SQL engine, where procedural statements are executed and SQL statements are sent to the SQL executor in the Oracle engine. Since the PL/SQL engine resides in the Oracle engine, this is an efficient and swift operation.

The call to the Oracle engine needs to be made only once to execute any number of SQL statements, if these SQL sentences are bundled inside a PL/SQL block.

Diagram 15.2 gives an idea of how these statements are executed and how convenient it is to bundle SQL code within a PL/SQL block. Since the Oracle engine is called only once for each block, the speed of SQL statement execution is vastly enhanced, when compared to the Oracle engine being called once for each SQL sentence.

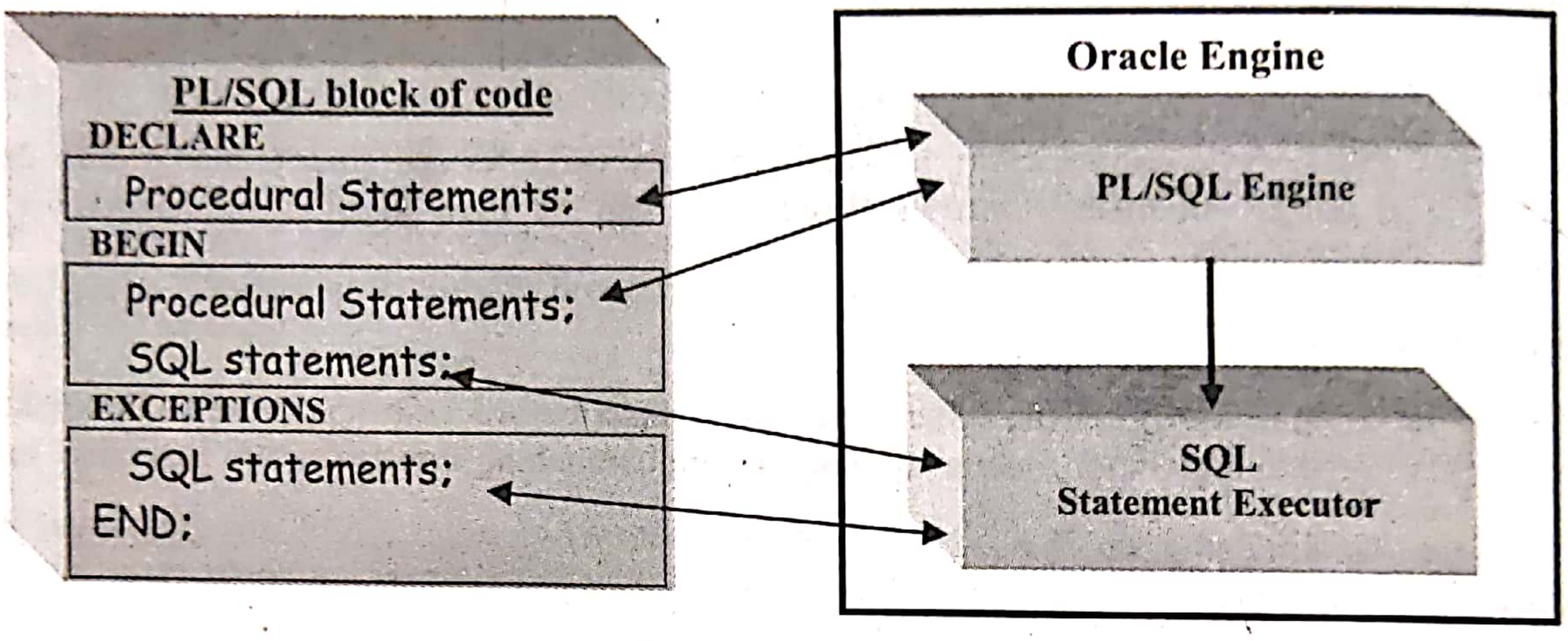

Diagram 15.2: The PL/SQL Execution Environment.

PL/SQL

The Character Set

The basic character set includes the following:

- Uppercase alphabets { **A - Z** }
- Lowercase alphabets { **a - z** }
- Numerals { **0 - 9** }
- Symbols **( ) + - * / < > = ! ; : . ' @ % , " # $ ^ & _ \ { } ? [ ]**

Words used in a PL/SQL block are called **Lexical Units**. Blank spaces can be freely inserted between lexical units in a PL/SQL block. The blank spaces have no effect on the PL/SQL block.

The ordinary symbols used in PL/SQL blocks are:

( ) + - * / < > = ; % ' " [ ] :

Compound symbols used in PL/SQL blocks are:

< > |= ~= ^= <= >= := ** .. || << >>

Literals

A literal is a numeric value or a character string used to represent itself.

Numeric Literal

These can be either integers or floats. If a float is being represented, then the integer part must be separated from the float part by a period.

Example:
25, 6.34, 7g2, 25e-03, .1, 1., 1.e4, +17, -5

String Literal

These are represented by one or more legal characters and must be enclosed within single quotes. The single quote character can be represented, by **writing it twice** in a string literal. This is **definitely not** the same as a double quote.

Example:
'Hello World', 'Don''t go without saving your work'

Character Literal

These are string literals consisting of single characters.

Example:
'*', 'A', 'Y'

Logical (Boolean) Literal

These are predetermined constants. The values that can be assigned to this data type are: TRUE, FALSE, NULL

PL/SQL Data Types

Both PL/SQL and Oracle have their foundations in SQL. Most PL/SQL data types are native to Oracle's data dictionary. Hence, there is a very easy integration of PL/SQL code with the Oracle Engine.

The default data types that can be declared in PL/SQL are **number** (for storing numeric data), **char** (for storing character data), **date** (for storing date and time data), **boolean** (for storing TRUE, FALSE or NULL). **number, char** and **date** data types can have NULL values.

The **%TYPE** attribute provides for further integration. PL/SQL can use the **%TYPE** attribute to declare variables based on definitions of columns in a table. Hence, if a column's attributes change, the variable's attributes will change as well. This provides for data independence, reduces maintenance costs, and allows programs to adapt to changes made to the table.

%TYPE declares a variable or constant to have the same data type as that of a previously defined variable or of a column in a table or in a view. When referencing a table, a user may name the table and the column, or name the owner, the table and the column.

NOT NULL causes creation of a variable or a constant that **cannot** be assigned a null value. If an attempt is made to assign the value NULL to a variable or a constant that has been assigned a **NOT NULL** constraint, Oracle senses the exception condition automatically and an internal error is returned.

Note

As soon as a variable or constant has been declared as NOT NULL, it must be assigned a value. Hence every variable or constant declared as NOT NULL **needs** to be followed by a PL/SQL expression that loads a value into the variable or constant.

Variables

Variables in PL/SQL blocks are **named** variables. A variable name must begin with a character and can be followed by a maximum of 29 other characters.

Reserved words cannot be used as variable names **unless** enclosed within double quotes. Variables must be separated from each other by at least one space or by a punctuation mark.

Case is insignificant when declaring variable names. A space cannot be used in a variable name. A variable of any data type either native to the Oracle Engine such as number, char, date, and so on or native to PL/SQL such as Boolean (i.e. logical variable content) can be declared.

Assigning Values to Variables

The assigning of a value to a variable can be done in two ways:

- Using the assignment operator := (i.e. a colon followed by an equal to sign).
- Selecting or fetching table data values into variables.

Note

An interesting point to note here is that if PL/SQL code blocks are used for loading and calculating variables, the power of the Oracle Engine is not used. This frees up the Oracle engine for other work and considerably improves response time.

Constants

Declaring a constant is similar to declaring a variable except that the keyword **constant** must be added to the variable name **and** a value assigned immediately. Thereafter, no further assignments to the constant are possible, while the constant is **within** the scope of the PL/SQL block.

Raw

Raw types are used to store **binary** data. Character variables are automatically converted between character sets by Oracle, if necessary. These are similar to char variables, except that they are not converted between character sets. It is used to store fixed length binary data. The maximum length of a raw variable is 32,767 bytes. However, the maximum length of a database raw column is 255 bytes.

Long raw is similar to long data, except that PL/SQL will not convert between character sets. The maximum length of a long raw variable is 32,760 bytes. The maximum length of a long raw column is 2 GB.

Rowid

This data type is the same as the database **ROWID** pseudo-column type. It can hold a rowid, which can be considered as a unique key for every row in the database. Rowids are stored internally as a fixed length binary quantity, whose actual fixed length varies depending on the operating system.

Various **DBMS_ROWID** functions are used to extract information about the ROWID pseudo-column. **Extended** and **Restricted** are two rowid formats. **Restricted** is used mostly to be backward compatible with previous versions of Oracle. The **Extended** format takes advantage of new Oracle features.

The **DBMS_ROWID** package has several procedures and functions to interpret the ROWIDs of records. The following table shows the **DBMS_ROWID** functions:

FUNCTION	DESCRIPTION
ROWID_VERIFY	Verifies if the ROWID can be extended; 0 = can be converted to extended format; 1 = cannot be converted to extended format
ROWID_TYPE	0 = ROWID, 1 = Extended
ROWID_BLOCK_NUMBER	The block number that contains the record; 1 = Extended ROWID
ROWID_OBJECT	The object number of the object that contains the record.
ROWID_RELATIVE_FNO	The relative file number contains the record.
ROWID_ROW_NUMBER	The row number of the record.
ROWID_TO_ABSOLUTE_FNO	The absolute file number; user need to input rowid_val, schema, and object; the absolute file number is returned.
ROWID_TO_EXTENDED	Converts the ROWID from Restricted to Extended; user need to input restr_rowid, schema, object; the extended number is returned.
ROWID_TO_RESTRICTED	Converts the ROWID from Extended to Restricted.

ROWID is a pseudo-column that has a unique value associated with each record of the database.

The **DBMS_ROWID** package is created by the **ORACLE_HOME/RDBMS/ADMIN/DBMSUTIL.SQL** script.

This script is automatically run when the Oracle instance is created.

LOB Types

A company may decide that some comments about each of its vendors must be stored along with their details. This must be stored along with all the other details that they have on a particular vendor. This can be done in Oracle with the help of LOB types.

The LOB types are used to store large objects. A large object can be either a binary or a character value upto 4 GB in size. Large objects can contain unstructured data, which is accessed more efficiently than long or long raw data, with fewer restrictions. LOB types are manipulated using the **DBMS_LOB** package. There are four types of LOBs:

- **BLOB (Binary LOB)** – This stores unstructured binary data upto 4 GB in length. A blob could contain video or picture information.
- **CLOB (Character LOB)** – This stores single byte characters upto 4 GB in length. This might be used to store documents.
- **BFILE (Binary File)** – This stores a pointer to read only binary data stored as an external file outside the database.

Of these LOBs, **BFILE** is an **external** to the database. Internal objects store a **locator** in the Large Object column of a table. **Locator** is a pointer that specifies the actual location of LOB stored outside the database. The LOB locator for **BFILE** is a pointer to the location of the binary file stored by the operating system. The **DBMS_LOB** package is used to manipulate LOBs. Oracle supports data integrity and concurrency for all the LOBs except **BFILE** as the data is stored outside the database.

Storage for LOB data

The area required to store the LOB data can be specified at the time of creation of the table that includes the LOB column. The create table command has a storage clause that specifies the storage characteristics for the table.

Syntax:

```
CREATE TABLE <TableName> (<ColumnName> <Datatype> <Size()>,
    <ColumnName> <Datatype> <Size()>, <ColumnName> CLOB,...);
```

Logical Comparisons

PL/SQL supports the comparison between variables and constants in SQL and PL/SQL statements. These comparisons, often called **Boolean expressions**, generally consist of simple expressions separated by relational operators (<, >, =, < >, >=, <=) that can be connected by logical operators (AND, OR, NOT). A Boolean expression will always evaluate to **TRUE**, **FALSE** or **NULL**.

Displaying User Messages On The VDU Screen

Programming tools require a method through which messages can be displayed on the VDU screen.

DBMS_OUTPUT is a package that includes a number of procedures and functions that accumulate information in a buffer so that it can be retrieved later. These functions can also be used to display messages.

PUT_LINE puts a piece of information in the package buffer followed by an end-of-line marker. It can also be used to display a message. **PUT_LINE** expects a single parameter of character data type. If used to display a message, it is the **message** string.

To display messages, the **SERVEROUTPUT** should be set to **ON**. **SERVEROUTPUT** is a SQL *PLUS environment parameter that displays the information passed as a parameter to the **PUT_LINE** function.

Syntax:

```
SET SERVEROUTPUT [ON/OFF]
```

Comments

A comment can have two forms, as:

- The comment line begins with a double hyphen (--). The entire line will be treated as a comment.
- The comment line begins with a slash followed by an asterisk (/*) till the occurrence of an asterisk followed by a slash (*/). All lines within are treated as comments. This form of specifying comments can be used to span across multiple lines. This technique can also be used to enclose a section of a PL/SQL block that temporarily needs to be isolated and ignored.

CONTROL STRUCTURE

The flow of control statements can be classified into the following categories:

- Conditional Control
- Iterative Control
- Sequential Control

Conditional Control

PL/SQL allows the use of an **IF** statement to control the execution of a block of code. In PL/SQL, the **IF - THEN - ELSIF - ELSE - END IF** construct in code blocks allow specifying certain conditions under which a specific block of code should be executed.

Syntax:

```
IF <Condition> THEN
      <Action>
ELSIF <Condition> THEN
      <Action>
ELSE
      <Action>
END IF;
```

Example 1:
Write a PL/SQL code block that will accept an account number from the user, check if the users balance is less than the minimum balance, only then deduct Rs.100/- from the balance. The process is fired on the ACCT_MSTR table.

```
DECLARE
/* Declaration of memory variables and constants to be used in the
Execution section.*/
    mCUR_BAL number(11,2);
    mACCT_NO varchar2(7);
    mFINE number(4) := 100;
    mMIN_BAL constant number(7,2) := 5000.00;
BEGIN
/* Accept the Account number from the user*/
    mACCT_NO := &mACCT_NO;
/* Retrieving the current balance from the ACCT_MSTR table where the
ACCT_NO in the table is equal to the mACCT_NO entered by the user.*/
    SELECT CURBAL INTO mCUR_BAL FROM ACCT_MSTR WHERE ACCT_NO=
mACCT_NO;
/* Checking if the resultant balance is less than the minimum balance
of Rs.5000. If the condition is satisfied an amount of Rs.100 is
deducted as a fine from the current balance of the corresponding
ACCT_NO.*/
    IF mCUR_BAL < mMIN_BAL THEN
        UPDATE ACCT_MSTR SET CURBAL = CURBAL - mFINE
            WHERE ACCT_NO = mACCT_NO;
    END IF;
END;
```

Output:

```
Enter value for macct_no: 'SB9'
old  11:    mACCT_NO := &mACCT_NO;
new  11:    mACCT_NO := 'SB9';

PL/SQL procedure successfully completed.
```

Iterative Control

Iterative control indicates the ability to repeat or skip sections of a code block. A **loop** marks a sequence of statements that has to be repeated. The keyword **loop** has to be placed before the first statemer in the sequence of statements to be repeated, while the keyword **end loop** is placed immediately afte he last statement in the sequence. Once a loop begins to execute, it will **go on forever**. Hence a c itional statement that controls the number of times a loop is executed **always accompanies** loops.

PL/SQL supports the following structures for iterative control:

Simple Loop

In simple loop, the key word **loop** should be placed before the first statement in the sequence and the keyword **end loop** should be written at the end of the sequence to end the loop.

Syntax:

```
Loop
    <Sequence of statements>
End loop;
```

Example 2:
Create a simple loop such that a message is displayed when a loop exceeds a particular value.

```
DECLARE
    i number := 0;
BEGIN
    LOOP
        i := i + 2;
        EXIT WHEN i > 10;
    END LOOP;
    dbms_output.put_line('Loop exited as the value of i has reached ' || to_char(i));
END;
```

Output:

```
Loop exited as the value of i has reached 12
PL/SQL procedure successfully completed.
```

The WHILE loop

Syntax:

```
WHILE <Condition>
LOOP
    <Action>
END LOOP;
```

Example 3:
Write a PL/SQL code block to 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 an empty table named **Areas**, consisting of two columns **Radius** and **Area**.

Table Name: Areas

RADIUS	AREA

Create the table AREAS as:

```
CREATE TABLE AREAS (RADIUS NUMBER(5), AREA NUMBER(14,2));

DECLARE
/* Declaration of memory variables and constants to be used in the
Execution section.*/
   pi constant number(4,2) := 3.14 ;
   radius number(5);
   area number(14,2);

BEGIN
/* Initialize the radius to 3, since calculations are required for
radius 3 to 7 */
   radius := 3;

/* Set a loop so that it fires till the radius value reaches 7 */
   WHILE RADIUS <= 7
   LOOP
   /* Area calculation for a circle */
      area := pi * power(radius,2);
   /* Insert the value for the radius and its corresponding area
   calculated in the table */
      INSERT INTO areas      VALUES (radius, area);

   /* Increment the value of the variable radius by 1 */
      radius := radius + 1;
   END LOOP;
END;
```

The above PL/SQL code block initializes a variable **radius** to hold the value of 3. The area calculations are required for the radius between 3 and 7. The value for area is calculated first with radius 3, and the radius and area are inserted into the table **Areas**. Now, the variable holding the value of radius is incremented by 1, i.e. it now holds the value 4. Since the code is held within a loop structure, the code continues to fire till the radius value reaches 7. Each time the value of radius and area is inserted into the areas table.

After the loop is completed the table will now hold the following:

Table Name: Areas

RADIUS	AREA
3	28.26
4	50.24
5	78.5
6	113.04
7	153.86

The FOR Loop

Syntax:

```
FOR variable IN [REVERSE] start..end
LOOP
     <Action>
END LOOP;
```

Note

The **variable** in the For Loop need not be declared. Also the increment value cannot be specified. The For Loop variable is **always incremented** by **1**.

Example 4:
Write a PL/SQL block of code for inverting a number 5639 to 9365.

```
DECLARE
/* Declaration of memory variables and constants to be used in the
Execution section.*/
    given_number varchar(5) := '5639';
    str_length number(2);
    inverted_number varchar(5);

BEGIN
/* Store the length of the given number */
    str_length := length(given_number);
/* Initialize the loop such that it repeats for the number of times
equal to the length of the given number. Also, since the number is
required to be inverted, the loop should consider the last number first
and store it i.e. in reverse order */
    FOR cntr IN REVERSE 1..str_length
    /* Variables used as counter in the for loop need not be declared
    i.e. cntr declaration is not required */
    LOOP
    /* The last digit of the number is obtained using the substr
    function, and stored in a variable, while retaining the previous
    digit stored in the variable*/
        inverted_number := inverted_number || substr(given_number, cntr, 1);
    END LOOP;

/* Display the initial number, as well as the inverted number, which is
stored in the variable on screen */
    dbms_output.put_line ('The Given number is ' || given_number );
    dbms_output.put_line ('The Inverted number is ' || inverted_number );
END;
```

Output:

```
The Given number is 5639
The Inverted number is 9365
```

The above PL/SQL code block stores the given number as well its length in two variables. Since the FOR loop is set to repeat till the length of the number is reached and in reverse order, the loop will fire 4 times beginning from the last digit i.e. **9**. This digit is obtained using the function **SUBSTR**, and stored in a variable. The loop now fires again to fetch and store the second last digit of the given number. This is appended to the last digit stored previously. This repeats till each digit of the number is obtained and stored.

Sequential Control

The GOTO Statement

The **GOTO** statement changes the **flow of control** within a PL/SQL block. This statement allows execution of a section of code, which is not in the normal flow of control. The entry point into such a block of code is marked using the tags **<<userdefined name>>**. The **GOTO** statement can then make use of this user-defined name to jump into that block of code for execution.

Syntax:

```
GOTO <codeblock name>;
```

Example 5:

Write a PL/SQL block of code to achieve the following: If there are no transactions taken place in the last 365 days then mark the account status as **inactive**, and then record the account number, the opening date and the type of account in the **INACTV_ACCT_MSTR** table.

Table Name: INACTV_ACCT_MSTR

ACCT_NO	OPNDT	TYPE

Create the table INACTV_ACCT_MSTR as:

```
CREATE TABLE INACTV_ACCT_MSTR (
    ACCT_NO VARCHAR2(10), OPNDT DATE, TYPE VARCHAR2(2));
```

```
DECLARE
/* Declaration of memory variables and constants to be used in the
Execution section.*/
   mACCT_NO VARCHAR2(10);
   mANS VARCHAR2(3);
   mOPNDT DATE;
   mTYPE VARCHAR2(2);
BEGIN
/* Accept the Account number from the user*/
   mACCT_NO := &mACCT_NO;
/* Fetch the account number into a variable */
   SELECT 'YES' INTO mANS FROM TRANS_MSTR WHERE ACCT_NO = mACCT_NO
      GROUP BY ACCT_NO HAVING MAX(SYSDATE - DT) >365;

/* If there are no transactions taken place in last 365 days the
execution control is transferred to a user labelled section of code,
labelled as mark_status in this example. */
   IF mANS = 'YES' THEN
      GOTO mark_status;
      ELSE
      dbms_output.put_line('Account number: ' || mACCT_NO || 'is active');
   END IF;

/* A labelled section of code which updates the STATUS of account
number held in the ACCT_MSTR table. Further the ACCT_NO, OPNDT and the
TYPE are inserted in to the table INACTV_ACCT_MSTR. */
   << mark_status>>
   UPDATE ACCT_MSTR SET STATUS = 'I' WHERE ACCT_NO = mACCT_NO;
```

```
    SELECT OPNDT, TYPE INTO mOPNDT, mTYPE
        FROM ACCT_MSTR WHERE ACCT_NO = mACCT_NO;
    INSERT INTO INACTV_ACCT_MSTR (ACCT_NO, OPNDT,TYPE)
        VALUES (mACCT_NO, mOPNDT, mTYPE);
    dbms_output.put_line(' Account number: '|| mACCT_NO || 'is marked as inactive');
END;
```

The PL/SQL code first fetches the Account number from the user into a variable **mACCT_NO**. It then verifies using an SQL statement, whether any transactions are performed within **last 365 days.**

If they are, then a message stating **"Account Number ____ is active"** is displayed.

But if there are no transactions performed in the last 365 days (i.e. 1 year) then a value **"YES"** is stored in a variable named **mANS.**

Based on the value held in this variable the **ACCT_MSTR table** is **updated** by setting the value held in the field **STATUS** to **I.**

This is followed by an insert statement, which inserts the account number the opening date and the type of that account in the **INACTV_ACCT_MSTR** table.

Finally a message stating **"Account Number ____ is marked as inactive"** is displayed.

SELF REVIEW QUESTIONS

FILL IN THE BLANKS

1. Each time an SQL statement is executed, a _____ is made to the engine's resources.
2. PL/SQL is a _____ language.
3. Code blocks start with a _____ section.
4. The _____ section deals with handling of errors that arise during execution of the data manipulation statements, which make up the PL/SQL code block.
5. The _____ section marks the end of a PL/SQL block.
6. Words used in a PL/SQL block are called _____ _____.
7. A _____ is a numeric value or a character string used to represent itself.
8. In a numeric literal, if a float is being represented, then the integer part must be separated from the float part by a _____.
9. _____ literals can be either integers or floats.
10. The _____ attribute is used to declare variables based on definitions of columns in a table.
11. Raw types are used to store _____ data.
12. The maximum length of a long raw column is _____.
13. _____ are stored internally as a fixed length binary quantity, whose actual fixed length varies depending on the operating system.
14. The _____ function verifies the block number that contains the record,

15. The _____ function verifies the absolute file number.
16. _____ data type stores unstructured binary data upto 4GB in length.
17. Internal objects store a _____ in the Large Object column of a table.
18. _____ is a pointer that specifies the actual location of LOB stored outside the database.
19. _____ puts a piece of information in the package buffer followed by an end-of-line marker.
20. The _____ function converts the ROWID from extended to restricted.
21. A _____ marks a sequence of statements that has to be repeated.
22. The _____ statement changes the flow of control within a PL/SQL block.

TRUE OR FALSE

23. SQL does not provide the programming techniques of conditional checking.
24. Multiple SQL statements are passed to the Oracle Engine at a time.
25. SQL has facility for programmed handling of errors that arise during manipulation of data.
26. A PL/SQL block has a definite structure which can be divided into sections.
27. Actual data manipulation, retrieval, looping and branching constructs are specified in the declare section.
28. Blank spaces can be freely inserted between lexical units in a PL/SQL block.
29. TRUE, FALSE, NULL cannot be assigned to Logical literals.
30. NOT NULL causes creation of a variable or a constant that cannot have a null value.
31. The String literals should not be enclosed within single quotes.
32. Reserved words can be used as variable names in PL/SQL.
33. The ROWID_TO_EXTENDED function converts the ROWID from restricted to extended
34. Raw is used to store ASCII data.
35. The ROWID_VERIFY function verifies if the ROWID can be extended.
36. The CLOB (Character LOB) data type stores single byte characters upto 4GB in length.
37. DBMS_PROC is a package that includes a number of procedures and functions that accumulate information in a buffer so that it can be retrieved later.
38. The DBMS_LOB package is used to manipulate LOBs.
39. The comment line begins with an asterisk followed by a slash.
40. The keyword loop has to be placed after the first statement in the sequence of statements to be repeated.
41. The variable in the FOR loop should always be declared.