INTRODUCTION

PL/SQL

- PL/SQL is Oracle's *procedural* language extension to SQL, the non-procedural relational database language.

- With PL/SQL, you can use SQL statements to manipulate ORACLE data and the *flow* of control statements to process the data. Moreover, you can declare constants and variables, define subprograms (procedures and functions), and trap runtime errors. Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

What is PL/SQL?

✓PL/SQL is a Procedural Language extension of **Structured Query Language (SQL)**.

✓PL/SQL is an extension of Structured Query Language (SQL) that is used in Oracle. Unlike SQL, PL/SQL allows the programmer to write code in a procedural format.

✓**PL/SQL (Procedural Language/Structured Query Language)** program executes on Oracle database.

✓PL/SQL is **specially designed** for Database oriented activities. Oracle PL/SQL allows you to perform data manipulation operation those are **safe and flexible.**

PL/SQL

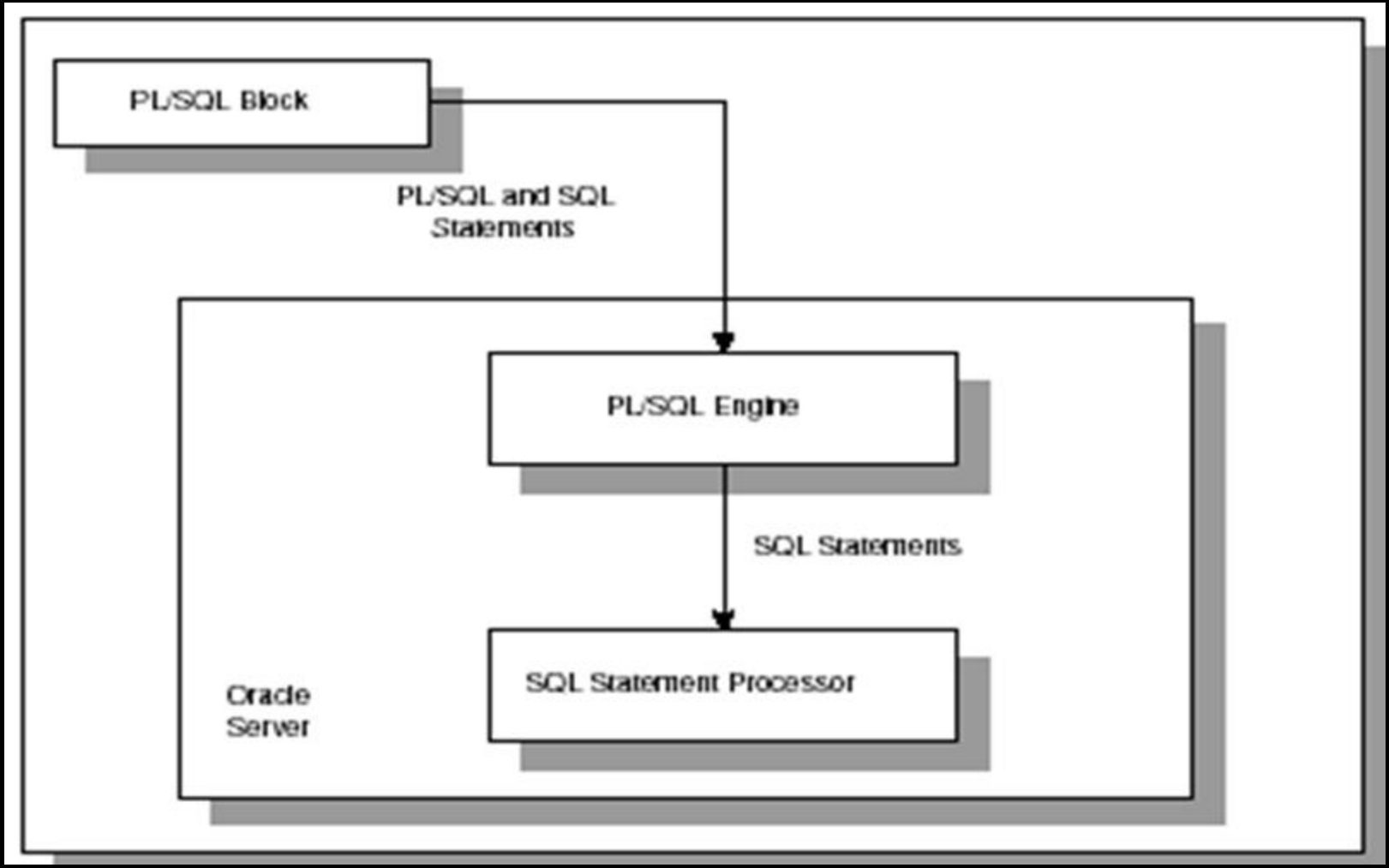
- ✓ It combines the data manipulation power of SQL with the processing power of procedural language to create super powerful SQL queries
- ✓ While PL/SQL is just like any other programming language, it has syntax and rules that determine how programming statements work together. It is important for you to realize that PL/SQL is not a stand-alone programming language.
- ✓ PL/SQL is a part of the Oracle RDBMS, and it can reside in two environments, the client and the server.

PL/SQL

- As a result, it is very easy to move PL/SQL modules between server-side and client-side applications.

- When the PL/SQL engine is located on the server, the whole PL/SQL block is passed to the PL/SQL engine on the Oracle server.

The PL/SQL engine processes the block according to the following Figure



Combines the flexibility of SQL (4GL) with the power and configurability of the procedural constructs of a 3GL.

Extends SQL by adding 3GL constructs such as:

- Variables and types (predefined and user defined)
- Control Structures (IF-THEN-ELSE, Loops)
- Procedures and functions
- Object types and methods

PL/SQL based on language constructs

- Block Structure
- Error Handling
- Variables and Types
- Conditionals
- Looping Constructs
- Cursors

PL/SQL

- When the PL/SQL engine is located on the client, as it is in the Oracle Developer Tools, the PL/SQL processing is done on the client side.
- All SQL statements that are embedded within the PL/SQL block are sent to the Oracle server for further processing. When PL/SQL block contains no SQL statement, the entire block is executed on the client side.

DIFFERENCE BETWEEN PL/SQL AND SQL

- When a SQL statement is issued on the client computer, the request is made to the database on the server, and the result set is sent back to the client.

- As a result, a single SQL statement causes two trips on the network. If multiple SELECT statements are issued, the network traffic increase significantly very fast. For example, four SELECT statements cause eight network trips.

- If these statements are part of the PL/SQL block, they are sent to the server as a single unit. The SQL statements in this PL/SQL program are executed at the server and the result set is sent back as a single unit. There is still only one network trip made as is in case of a single SELECT statement.

Comparison of SQL*PLUS and PL/SQL

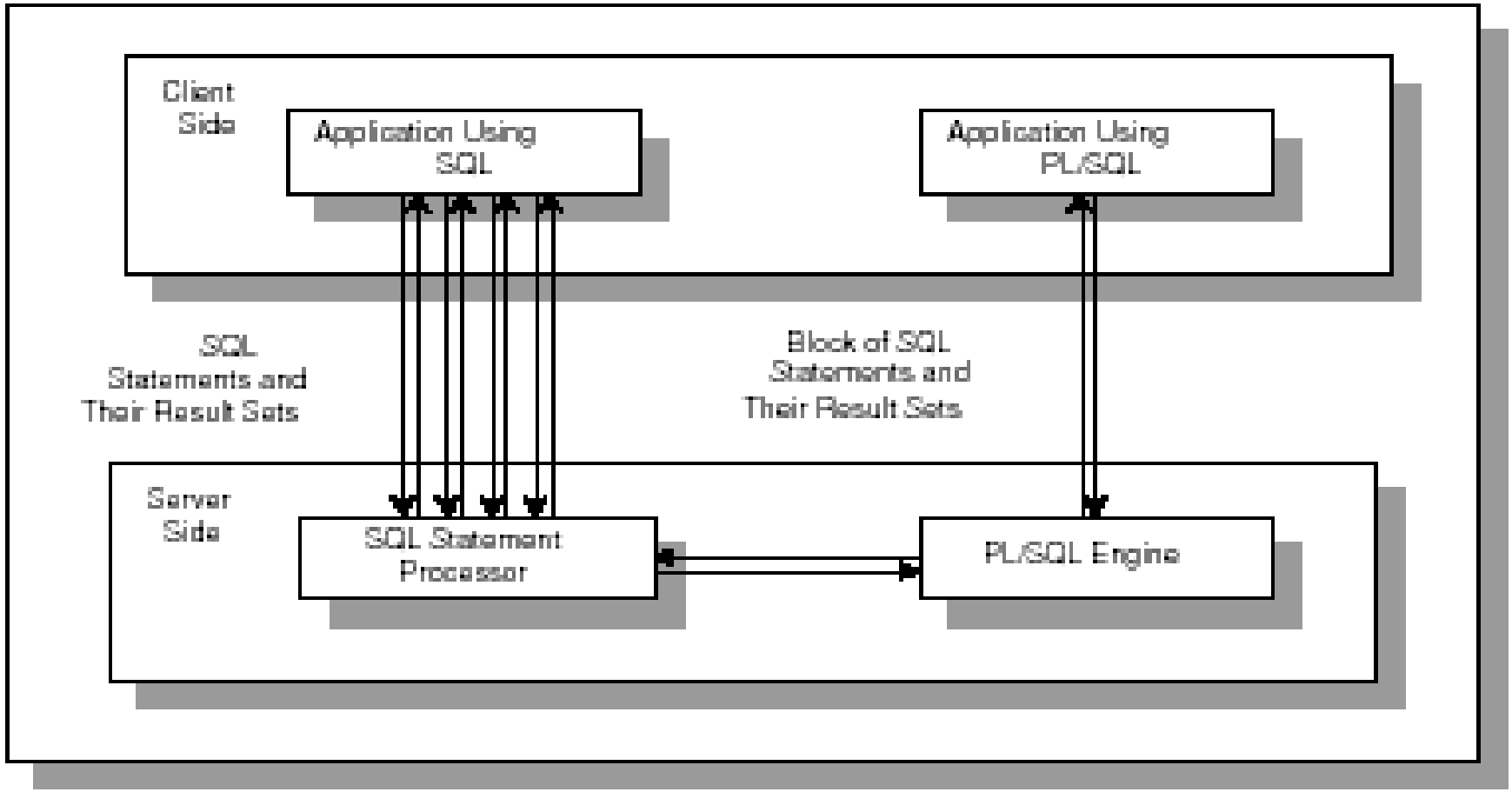


Figure 2.2 ■ PL/SQL in client-server architecture.

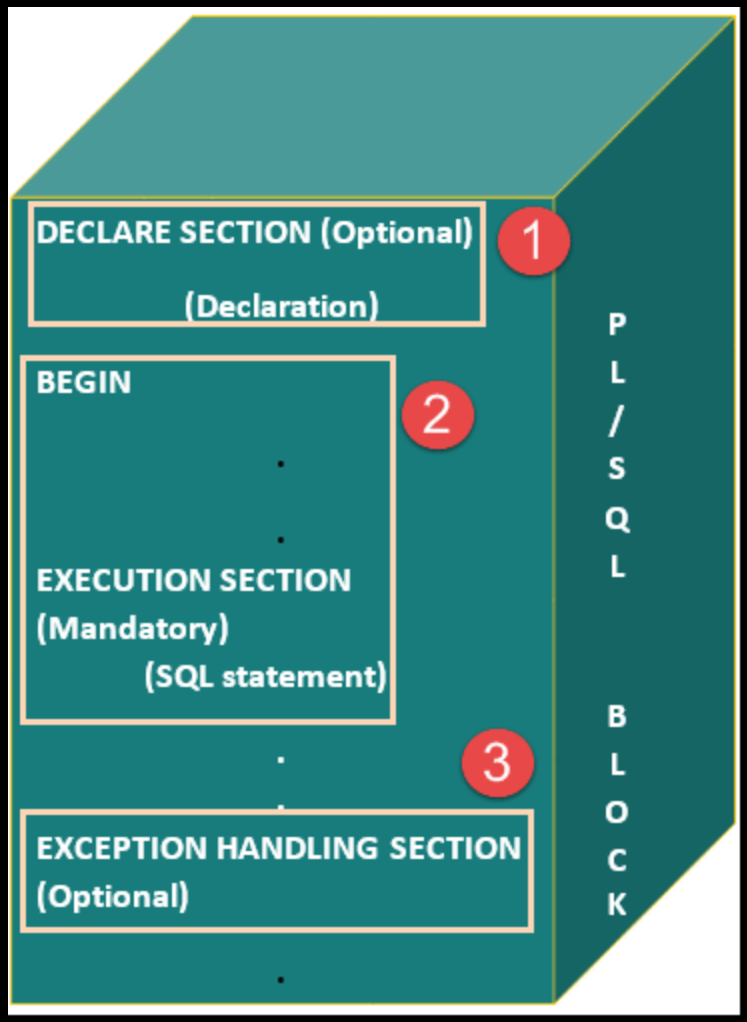
Fundamentals of PL/SQL

- Full-featured programming language
- Interpreted language
- Execute using Oracle 10*g* utilities
  - SQL*Plus
  - Forms Builder
- Combines SQL queries with procedural commands
- Reserved words

Item Type	Capitalization	Example
Reserved word	Uppercase	BEGIN, DECLARE
Built-in function	Uppercase	COUNT, TO_DATE
Predefined data type	Uppercase	VARCHAR2, NUMBER
SQL command	Uppercase	SELECT, INSERT
Database object	Lowercase	student, f_id
Variable name	Lowercase	current_s_id, current_f_last

Table 4-1 PL/SQL command capitalization styles

PL/SQL BLOCKS



PL/SQL Program Blocks

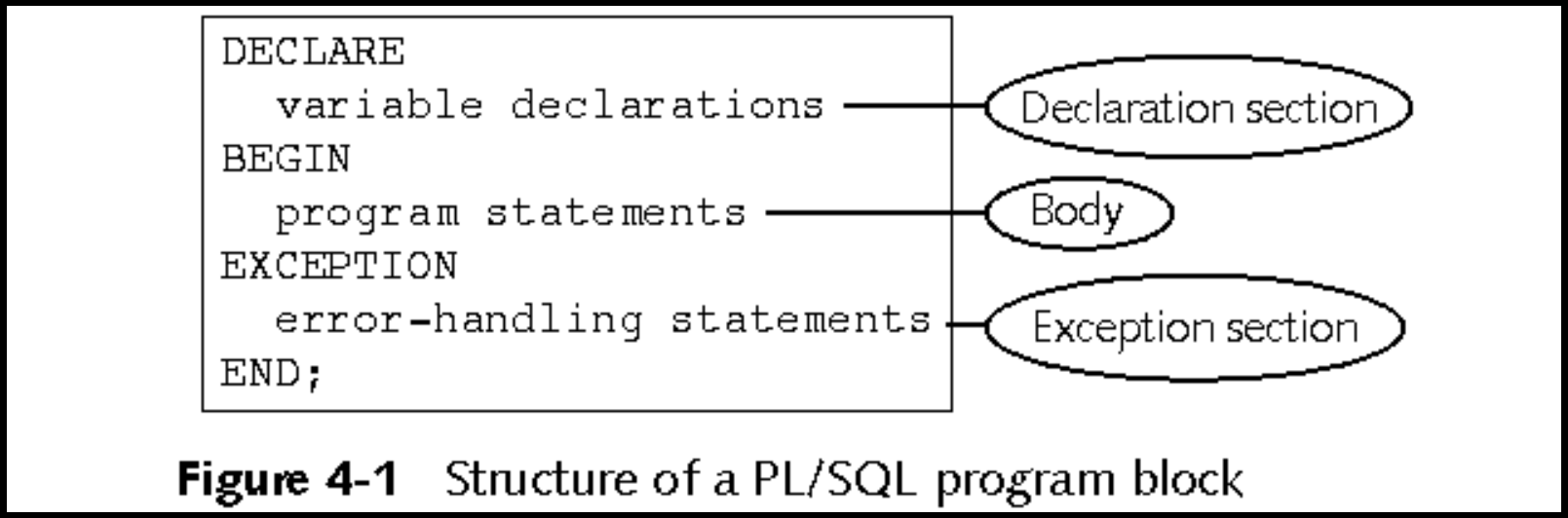


Figure 4-1 Structure of a PL/SQL program block

- Comments:
  - Not executed by interpreter
  - Enclosed between /* and */
  - On one line beginning with --

PL/SQL BLOCKS

- PL/SQL blocks can be divided into two groups:
  1. Named
  2. Anonymous.
- Named blocks are used when creating subroutines. These subroutines, are procedures, functions, and packages.
- The subroutines can be stored in the database and referenced by their names later on.
- In addition, subroutines can be defined within the anonymous PL/SQL block.
- Anonymous PL/SQL blocks do not have names. As a result,they cannot be stored in the database and referenced later.

PL/SQL BLOCK STRUCTURE

- PL/SQL blocks contain three sections
  1. Declare section
  2. Executable section and
  3. Exception-handling section.

- The executable section is the only mandatory section of the block.
- Both the declaration and exception-handling sections are optional.

PL/SQL Arithmetic Operators in describing Order of Precedence

Operator	Description	Example	Result
**	Exponentiation	2 ** 3	8
* /	Multiplication Division	2 * 3 9/2	6 4.5
+ –	Addition Subtraction	3 + 2 3 – 2	5 1
–	Negation	–5	–5

Table 4-5 PL/SQL arithmetic operators in describing order of precedence

- Parentheses are used to force PL/SQL interpreter to evaluate operations in a certain order

```
total_hours_worked - 40 * over_time_rate

(total_hours_worked – 40) * over_time_rate
```

Assignment Statements

```
DECLARE
     variable1 NUMBER := 10;
     variable2 NUMBER := 0;
BEGIN
     variable2 := variable1 +1;
END;
```

- Assigns value to variable
- Operator: **:=**
- Syntax: *variable_name* := *value*;
- String literal within single quotation mark
- Examples:

```
current_s_first_name := 'Gautam';
current_student_ID NUMBER := 100;
```

- Result of adding a value to a NULL value is another NULL value
- DEFAULT keyword can be used instead of assignment operator

Q: What is the final value of variable2?

Displaying PL/SQL Program Output in SQL*Plus

- PL/SQL output buffer
  - Memory area on database server
  - Stores program's output values before they are displayed to user
  - Default buffer size is 2000 bytes
  - Should increase size if you want to display more than a few lines in SQL Plus to avoid buffer overflow error
  - Syntax: `SET SERVEROUTPUT ON SIZE buffer_size`
  - Example: `SET SERVEROUTPUT ON SIZE 4000`

Displaying PL/SQL Program Output in SQL*Plus

- DBMS_OUTPUT
  - is an Oracle built-in package
  - Consists of a set of programs for processing output
- PUT_LINE is the DBMS_OUTPUT procedure for displaying output
  - Syntax:
    ```
    DBMS_OUTPUT.PUT_LINE('display_text');
    ```
  - Example:
    ```
    DBMS_OUTPUT.PUT_LINE(current_s_first);
    ```
  - Displays maximum of 255 characters of text data
  - If try to display more than 255 characters, error occurs

Writing a PL/SQL Program

- Write PL/SQL program in Notepad or another text editor
- Indenting commands within each section is a good programming practice.
- Copy and paste program commands from text editor into SQL*Plus
- Press Enter after last program command
- Type front slash (/)
- Then press Enter again

PL/SQL Program Commands

```
--PL/SQL program to display the current date
DECLARE
   todays_date DATE;
BEGIN
   todays_date := SYSDATE;
   DBMS_OUTPUT.PUT_LINE('Today''s date is ');
   DBMS_OUTPUT.PUT_LINE(todays_date);
END;
```

Figure 4-3 PL/SQL program commands

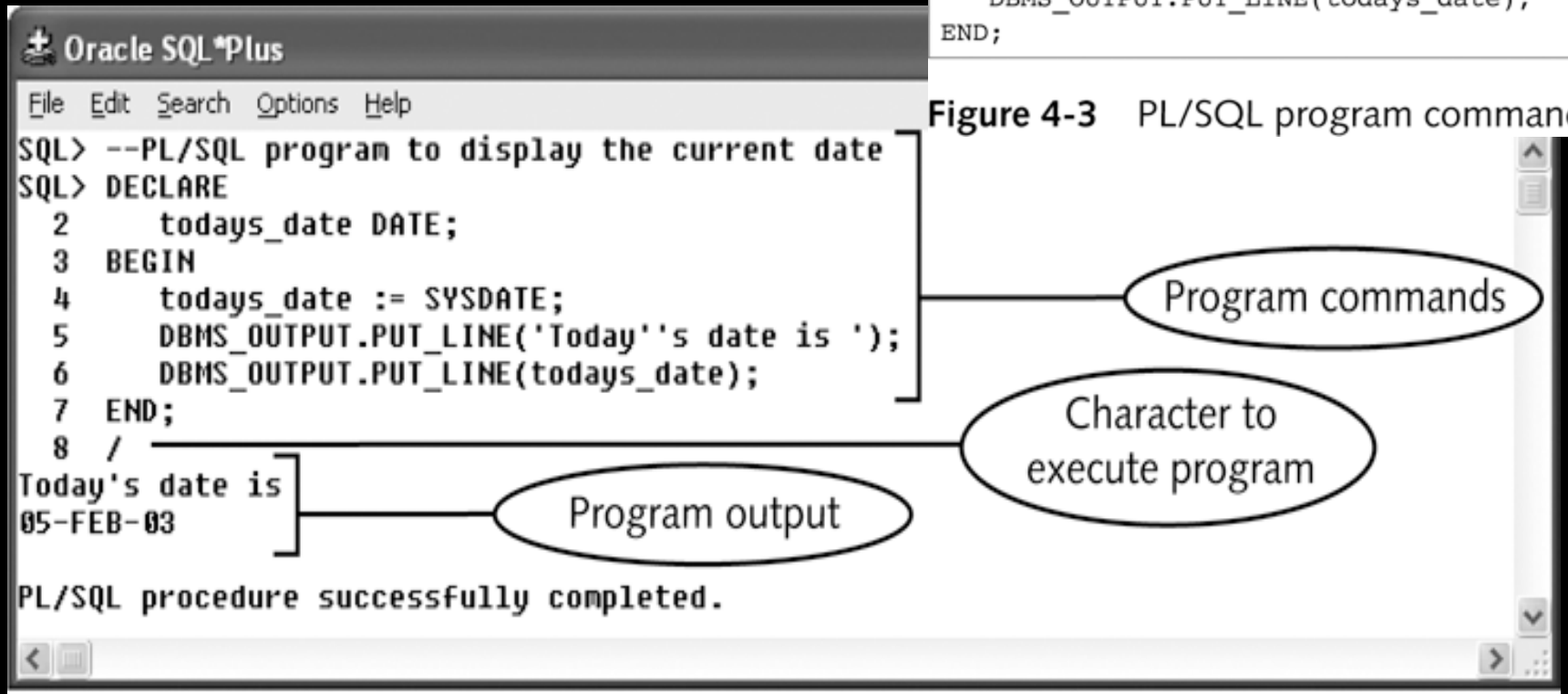


Figure 4-4 Executing a PL/SQL program in SQL*Plus

PL/SQL Data Conversion Functions

```
WHERE O_DATE = TO_DATE ('29/05/2006', 'DD/MM/YYYY')
```

```
WHERE O_DATE = '29/05/2006'
```

- Implicit data conversions
  - Interpreter automatically converts value from one data type to another
  - If PL/SQL interpreter unable to implicitly convert value error occurs
- Explicit data conversions
  - Convert variables to different data types
  - Using data conversion functions

Data Conversion Function	Description	Example
TO_CHAR	Converts either a number or a date value to a string using a specific format model	`TO_CHAR(2.98, '$999.99');` `TO_CHAR(SYSDATE, 'MM/DD/YYYY');`
TO_DATE	Converts a string to a date using a specific format model	`TO_DATE('07/14/2003', 'MM/DD/YYYY');`
TO_NUMBER	Converts a string to a number	`TO_NUMBER('2');`

Table 4-6 PL/SQL data conversion functions of PL/SQL

Manipulating Character Strings

- **Concatenating**
  - Joining two separate strings
  - Operator: `||` (i.e. double bar)
  - Syntax: *`new_string := string1 || string2;`*
  - Example: `s_fullname := s_first || s_last;`
- Parse
  - Separate single string consisting of two data items separated by commas or spaces

```
s_fullname := s_first ||' '|| s_last;
```

Variable	Data type	Value
Bldg_code	VARCHAR2	LH
Room_num	VARCHAR2	101
Room_capacity	NUMBER	150

```
room_message := bldg_code || ' Room ' || room_num
|| ' has ' || TO_CHAR(room_capacity) || 'seats.';
```

23

Question: Write down the value of `room_message` after the above Assignment statement is executed.

Debugging PL/SQL Programs

- Syntax error
  - Occurs when command does not follow guidelines of programming language
  - Generate compiler or interpreter error messages
- Logic error
  - Does not stop program from running
  - Results in incorrect result

Program with a Syntax Error

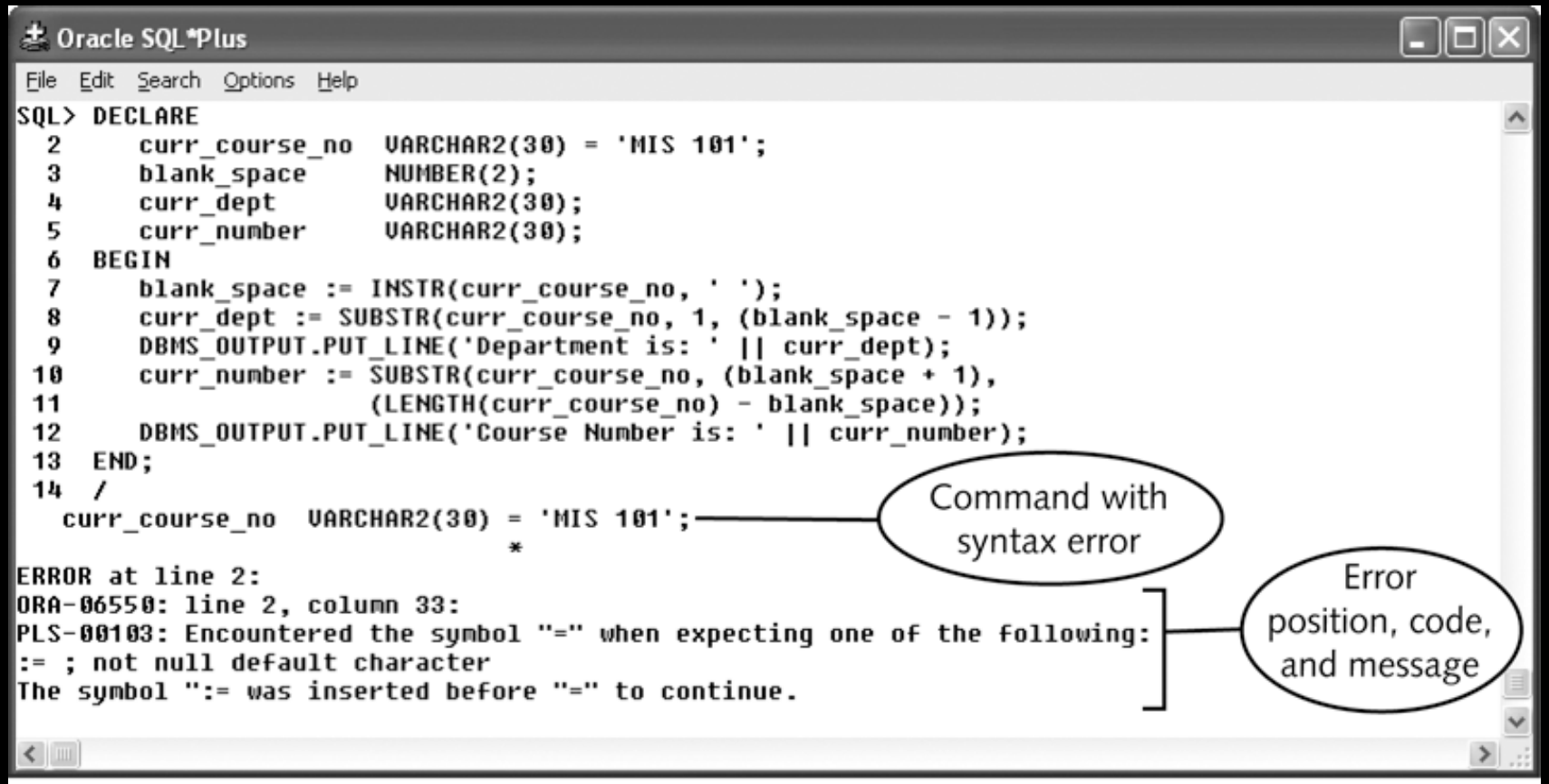


Figure 4-8 Program with a syntax error

Program with a Logic Error

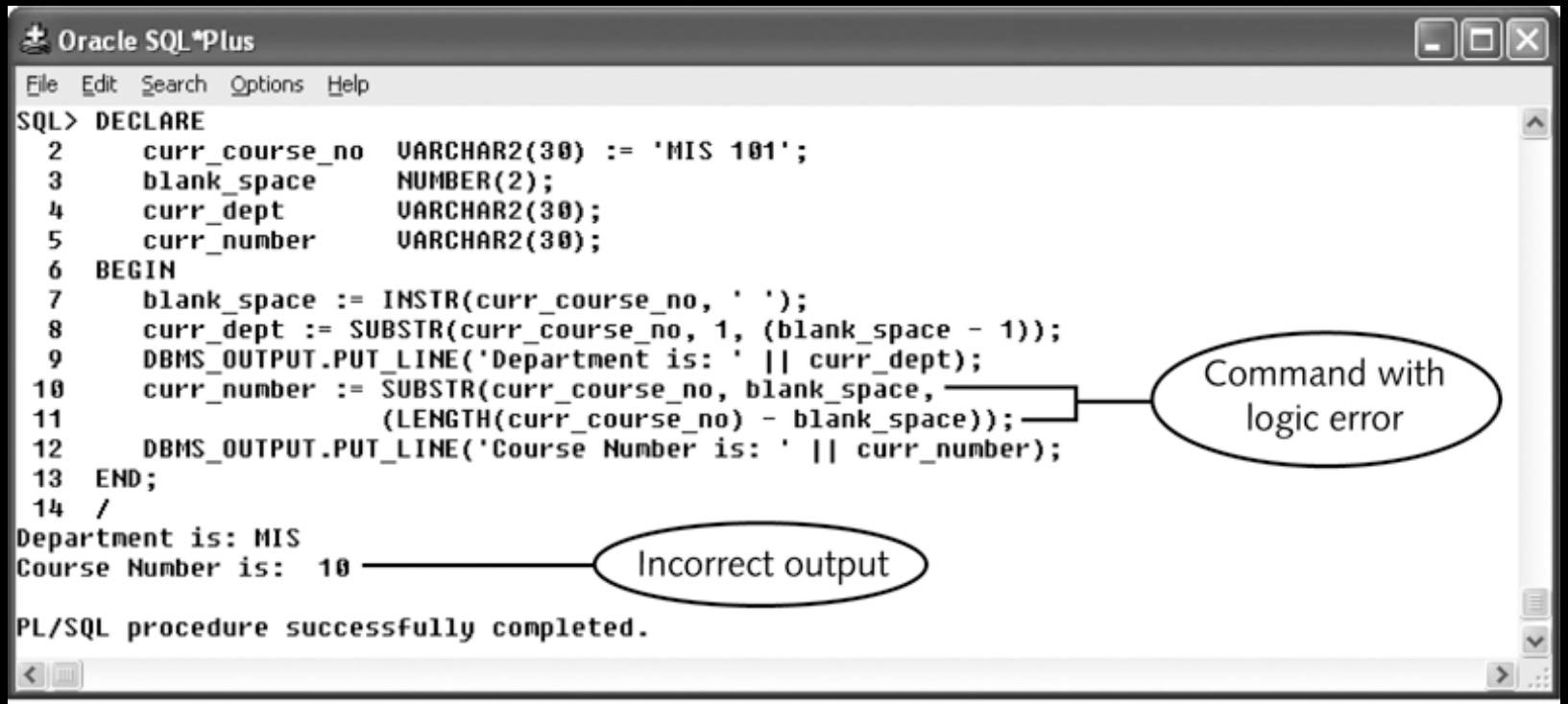


Figure 4-9 Program with a logic error

- Which of the following is the source of the error?
  - `LENGTH(curr_course_no) - blank_space));`
  - `SUBSTR(curr_course_no, blank_space,`

PL/SQL BLOCK STRUCTURE

- PL/SQL block has the following structure:

```
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCETION
    Exception-handling statements
END ;
```

DECLARATION SECTION

- The *declaration section* is the first section of the PL/SQL block.
- It contains definitions of PL/SQL identifiers such as variables, constants, cursors and so on.
- Example

```
DECLARE
    v_first_name VARCHAR2(35) ;
    v_last_name VARCHAR2(35) ;
    v_counter NUMBER := 0 ;
```

EXECUTABLE SECTION

- The executable section is the next section of the PL/SQL block.
- This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section.

```
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
        WHERE student_id = 123 ;
    DBMS_OUTPUT.PUT_LINE
    ('Student name :' || v_first_name ||'  '||
    v_last_name);
END;
```

EXCEPTION-HANDLING SECTION

- The *exception-handling section* is the last section of the PL/SQL block.
- This section contains statements that are executed when a runtime error occurs within a block.
- Runtime errors occur while the program is running and cannot be detected by the PL/SQL compiler.

```
EXCEPTION
        WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE
          (‘ There is no student with student id 123 ’);
END;
```

HOW PL/SQL GETS EXECUTED

- Every time an anonymous block is executed, the code is sent to the PL/SQL engine on the server where it is compiled.
- The named PL/SQL block is compiled only at the time of its creation, or if it has been changed.
- The compilation process includes syntax checking, binding and p-code generation.
- Syntax checking involves checking PL/SQL code for syntax or compilation errors.
- Once the programmer corrects syntax errors, the compiler can assign a storage address to program variables that are used to hold data for Oracle. This process is called ***Binding.***

HOW PL/SQL GETS EXECUTED

- After binding, p-code is generated for the PL/SQL block.
- P-code is a list of instructions to the PL/SQL engine.
- For named blocks, p-code is stored in the database, and it is used the next time the program is executed.
- Once the process of compilation has completed successfully, the status for a named PL/SQL block is set to VALID, and also stored in the database.
- If the compilation process was not successful, the status for a named PL/SQL block is set to INVALID.

PL/SQL IN SQL*PLUS

- SQL*Plus is an interactive tool that allows you to type SQL or PL/SQL statements at the command prompt.
- These statements are then sent to the database. Once they are processed, the results are sent back from the database and displayed on the screen.
- There are some differences between entering SQL and PL/SQL statements.

SQL EXAMPLE

```
SELECT first_name, last_name
FROM student
WHERE student_id = 123;
```

- The semicolon terminates this SELECT statement. Therefore, as soon as you type semicolon and hit the ENTER key, the result set is displayed to you.

PL/SQL EXAMPLE

```
DECLARE
        v_first_name VARCHAR2(35);
        v_last_name VARCHAR2(35);
BEGIN
        SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
        WHERE student_id = 123;
        DBMS_OUTPUT.PUT_LINE
        ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE
                        ('There is no student with student id 123');
END;
.
/
```

PL/SQL EXAMPLE

- There are two additional lines at the end of the block containing “.” and “/”. The “.” marks the end of the PL/SQL block and is optional.
- The “/” executes the PL/SQL block and is required.
- When SQL*Plus reads SQL statement, it knows that the semicolon marks the end of the statement. Therefore, the statement is complete and can be sent to the database.
- When SQL*Plus reads a PL/SQL block, a semicolon marks the end of the individual statement within the block. In other words, it is not a block terminator.

PL/SQL EXAMPLE

- Therefore, SQL*Plus needs to know when the block has ended. As you have seen in the example, it can be done with period and forward slash.

```
.
/
```

EXECUTING PL/SQL

- ✓ PL/SQL can be executed directly in SQL*Plus.
- ✓ A PL/SQL program is normally saved with an .sql extension.
- ✓ To execute an anonymous PL/SQL program, simply type the following command at the SQL prompt:

```
SQL> @DisplayAge
```

GENERATING OUTPUT

- Like other programming languages, PL/SQL provides a procedure (i.e. PUT_LINE) to allow the user to display the output on the screen. For a user to able to view a result on the screen, two steps are required.
- First, before executing any PL/SQL program, type the following command at the SQL prompt

  (Note: you need to type in this command only once for every SQL*PLUS session):

  SQL> **SET SERVEROUTPUT ON**;

  or put the command at the beginning of the program, right before the declaration section.

GENERATING OUTPUT

Second, use **DBMS_OUTPUT.PUT_LINE** in your executable section to display any message you want to the screen.

Syntax for displaying a message:

DBMS_OUTPUT.PUT_LINE(<string>);

in which PUT_LINE is the procedure to generate the output on the screen, and DBMS_OUTPUT is the package to which the PUT_LINE belongs.

DBMS_OUTPUT_PUT_LINE('My age is ' || num_age);

SUBSTITUTION VARIABLES

- SQL*Plus allows a PL/SQL block to receive input information with the help of substitution variables.
- Substitution variables cannot be used to output the values because no memory is allocated for them.
- SQL*Plus will substitute a variable before the PL/SQL block is sent to the database.
- Substitution variables are usually prefixed by the ampersand(&) character or double ampersand (&&) character.

EXAMPLE

```
DECLARE
        v_student_id NUMBER := &sv_student_id;
        v_first_name VARCHAR2(35);
        v_last_name VARCHAR2(35);
BEGIN
        SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
        WHERE student_id = v_student_id;
        DBMS_OUTPUT.PUT_LINE
                ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
        WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('There is no such student');
END;
```

EXAMPLE

- When this example is executed, the user is asked to provide a value for the student ID.
- The example shown above uses a single ampersand for the substitution variable.
- When a single ampersand is used throughout the PL/SQL block, the user is asked to provide a value for each occurrence of the substitution variable.

EXAMPLE

```
BEGIN
        DBMS_OUTPUT.PUT_LINE('Today is ‘||’&sv_day');
        DBMS_OUTPUT.PUT_LINE('Tomorrow will be ‘||’ &sv_day');
END;
```

This example produces the following output:

Enter value for sv_day: Monday
old 2: DBMS_OUTPUT.PUT_LINE('Today is ‘||’ &sv_day');
new 2: DBMS_OUTPUT.PUT_LINE('Today is ‘||’ Monday');
Enter value for sv_day: Tuesday
old 3: DBMS_OUTPUT.PUT_LINE('Tomorrow will be ‘||’ &sv_day');
new 3: DBMS_OUTPUT.PUT_LINE('Tomorrow will be ‘||’ Tuesday');
Today is Monday
Tomorrow will be Tuesday
PL/SQL procedure successfully completed.

EXAMPLE

- When a substitution variable is used in the script, the output produced by the program contains the statements that show how the substitution was done.
- If you do not want to see these lines displayed in the output produced by the script, use the SET command option before you run the script as shown below:

```
SET VERIFY OFF;
```

EXAMPLE

- Then, the output changes as shown below:

Enter value for sv_day: Monday
Enter value for sv_day: Tuesday
Today is Monday
Tomorrow will be Tuesday
PL/SQL procedure successfully completed.

- The substitution variable sv_day appears twice in this PL/SQL block. As a result, when this example is run, the user is asked twice to provide the value for the same variable.

EXAMPLE

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Today is '||'&&sv_day');
    DBMS_OUTPUT.PUT_LINE('Tomorrow will be '||'
    &sv_day');
END;
```

- In this example, substitution variable sv_day is prefixed by double ampersand in the first DBMS_OUTPUT.PUT_LINE statement.
- As a result, this version of the example produces different output.

OUTPUT

```
Enter value for sv_day: Monday
old 2: DBMS_OUTPUT.PUT_LINE('Today is '||' &&sv_day');
new 2: DBMS_OUTPUT.PUT_LINE('Today is '||' Monday');
old 3: DBMS_OUTPUT.PUT_LINE('Tomorrow will be '||' &sv_day');
new 3: DBMS_OUTPUT.PUT_LINE('Tomorrow will be '||' Monday');

Today is Monday
Tomorrow will be Monday
PL/SQL procedure successfully completed.
```

- It is clear that the user is asked only once to provide the value for the substitution variable sv_day.
- As a result, both DBMS_OUTPUT.PUT_LINE statements use the value of Monday entered previously by the user.

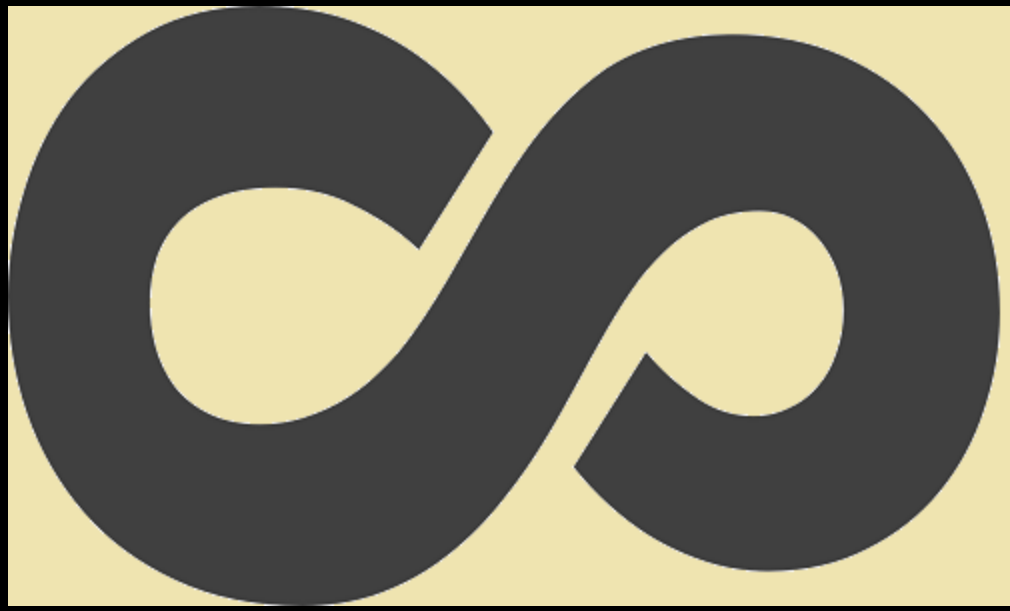
Substitution Variables

- Ampersand(&) character and double ampersand (&&) characters are the default characters that denote substitution variables.
- There is a special SET command option available in SQL*Plus that allows to change the default character (&) to any other character or disable the substitution variable feature.
- This SET command has the following syntax:

SET DEFINE *character*
or
SET DEFINE ON
or
SET DEFINE OFF

Substitution Variables

- The first set command option changes the prefix of the substitution variable from an ampersand to another character. This character cannot be alphanumeric or white space.
- The second (ON option) and third (OFF option) control whether SQL*Plus will look for substitution variables or not.
- In addition, ON option changes the value of the *character* back to the ampersand.



PL/SQL Loops

PL/SQL Loop：Basic Loop, FOR Loop, WHILE Loop

PL/SQL Loop Basic Loop, FOR Loop, WHILE Loop repeat a number of block statements in your PL/SQL program. Loop use when we have a block of statements for required to repeatedly certain number of times. PL/SQL loop statements 3 different forms:

1.Basic LOOP
2.WHILE LOOP
3.FOR LOOP

Oracle recommended to write a label when use loop statement. It's benefit to improve readability. label is not compulsory for execute loop. compiler does not check to label defined or not.

Define label before LOOP keyword and after END LOOP keyword.

Basic LOOP

Basic LOOP syntax:

```
[ label_name ] LOOP
   statement(s);
END LOOP [ label_name ];
```

Example:

```
DECLARE
no NUMBER := 5;
BEGIN
LOOP
DBMS_OUTPUT.PUT_LINE ('Inside value: no = ' || no);
no := no -1;
IF no = 0 THEN EXIT;
END IF;
END LOOP;
DBMS_OUTPUT.PUT_LINE('Outside loop end');
END;
/
```

WHILE LOOP

WHILE LOOP write in following syntax format:
[label_name] WHILE condition LOOP
  statement(s);
END LOOP [label_name];

EXAMPLE:

```
DECLARE
  no NUMBER := 0;

BEGIN
        WHILE no < 10 LOOP
        no := no + 1;
        END LOOP;
   DBMS_OUTPUT.PUT_LINE('Sum :' || sum);
END;
/
```

FOR LOOP

FOR LOOP write in following syntax format:
[label_name] FOR current_value IN [REVERSE]
lower_value..upper_value LOOP
   statement(s);
END LOOP [label_name];123

Example Code:

```
BEGIN
        FOR no IN 1 .. 5 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration : ' || no);
        END LOOP;
END;
```

REVERSE FOR Loop

Optional REVERSE keyword introduce to iteration is proceed from upper_value to lower_value range.

Example Code :

```
BEGIN
        FOR no IN REVERSE 5 .. 1 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration : ' || no);
        END LOOP;
END;
/
```

EXIT Statement

EXIT statement unconditionally exit the current loop iteration and transfer control to end of current loop. EXIT statement writing syntax,

```
DECLARE
  no NUMBER := 5;
BEGIN
        LOOP
        DBMS_OUTPUT.PUT_LINE ('Inside value:  no = ' || no);
        no := no -1;
        IF no = 0 THEN
                EXIT;
        END IF;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Outside loop end');
                        -- After EXIT control transfer this statement
END;
/
```

EXIT WHEN Statement

EXIT WHEN statement unconditionally exit the current loop iteration when WHEN clause condition true.

```
DECLARE
        i number;
BEGIN
        LOOP
                dbms_output.put_line('Hello');
                i:=i+1;
                EXIT WHEN i>5;
        END LOOP;
END;
/
```

CONTINUE Statement

CONTINUE Statement unconditionally skip the current loop iteration and next iteration iterate as normal, only skip matched condition.

Syntax:

```
IF condition THEN
        CONTINUE;
END IF;
```

Example Code:

```
DECLARE
 no NUMBER := 0;
BEGIN
FOR no IN 1 .. 5 LOOP
  IF i = 4 THEN
CONTINUE;
END IF;
 DBMS_OUTPUT.PUT_LINE('Iteration : ' || no);
END LOOP;
END;
/
```

CONTINUE WHEN Statement

CONTINUE WHEN Statement unconditionally skip the current loop iteration when WHEN clauses condition true,

Syntax:

```
CONTINUE WHEN condition;
        statement(s);
```

Example Code:

```
DECLARE
  no NUMBER := 0;
BEGIN
        FOR no IN 1 .. 5 LOOP
           DBMS_OUTPUT.PUT_LINE('Iteration : ' || no);
                CONTINUE WHEN no = 4
DBMS_OUTPUT.PUT_LINE('CONTINUE WHEN EXECUTE Iteration : ' || no);
        END LOOP;
END;
/
```

GOTO Statement

GOTO statement unconditionally transfer program control. GOTO statement writing syntax,

Syntax

GOTO code_name

```
----------
----------
```

<<code_name>>

Example:

```
BEGIN
FOR i IN 1..5 LOOP
dbms_output.put_line(i);
IF i=4 THEN
        GOTO label1;
END IF;
END LOOP;
<<label1>>
DBMS_OUTPUT.PUT_LINE('khela sesh!!');
END;
/
```

PL/SQL Case Statement

PL/SQL CASE statement comparing one by one sequencing conditions. CASE statement attempt to match expression that is specified in one or more WHEN condition.
CASE statement are following two forms:

1.Simple CASE Statement
2.Searched CASE Statement

Simple CASE Statement

PL/SQL simple CASE statement evaluates selector and attempt to match one or more WHEN condition.

```
CASE selector
        WHEN value-1
                THEN statement-1;
        WHEN value-2
                THEN statement-2;
        ELSE
                statement-3;
END CASE
```

```
DECLARE
        a number := 7;
BEGIN
        CASE a
                WHEN 1 THEN
                        DBMS_OUTPUT.PUT_LINE('value 1');
                WHEN 2 THEN
                        DBMS_OUTPUT.PUT_LINE('value 2');
                WHEN 3 THEN
                        DBMS_OUTPUT.PUT_LINE('value 3');
                        ELSE
                DBMS_OUTPUT.PUT_LINE('no matching CASE
found');
        END CASE;
END;
/
```

Searched CASE Statement

PL/SQL searched CASE statement has not selector and attempt to match one or more WHEN clauses condition.

Syntax

```
CASE
        WHEN condition-1 THEN
                statement-1;
        WHEN condition-2 THEN
                statement-2;
        ELSE
                statement-3;
END CASE;
```

```
DECLARE
        a number := 3;
BEGIN
        CASE
                WHEN a = 1 THEN
                        DBMS_OUTPUT.PUT_LINE('value 1');

                WHEN a = 2 THEN
                        DBMS_OUTPUT.PUT_LINE('value 2');

                WHEN a = 3 THEN
                        DBMS_OUTPUT.PUT_LINE('value 3');

                ELSE
        DBMS_OUTPUT.PUT_LINE('no matching CASE found');

        END CASE;
END;
/
```