PL/SQL

Overview
Data Types, Constant, Variables
Operators
Conditions & Loops
Cursors & Records
Arrays
Procedures & Functions
Triggers
Exception Handling
Packages

PL/SQL-Overview

What is PL/SQL?

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

PL/SQL is one of three key programming languages embedded in the Oracle Database,along with SQL itself and Java.

PL/SQL-Overview

PL/SQL stands for Procedural Language Extension to SQL.

PL/SQL extends SQL by adding programming structures and subroutines available in any high-level language.

PL/SQL is used for both server-side and client-side development.

Features

PL/SQL is language. A completely portable, high-performance Transaction-processing.

PL/SQL provides a built-in, interpreted and OS independent programming environment.

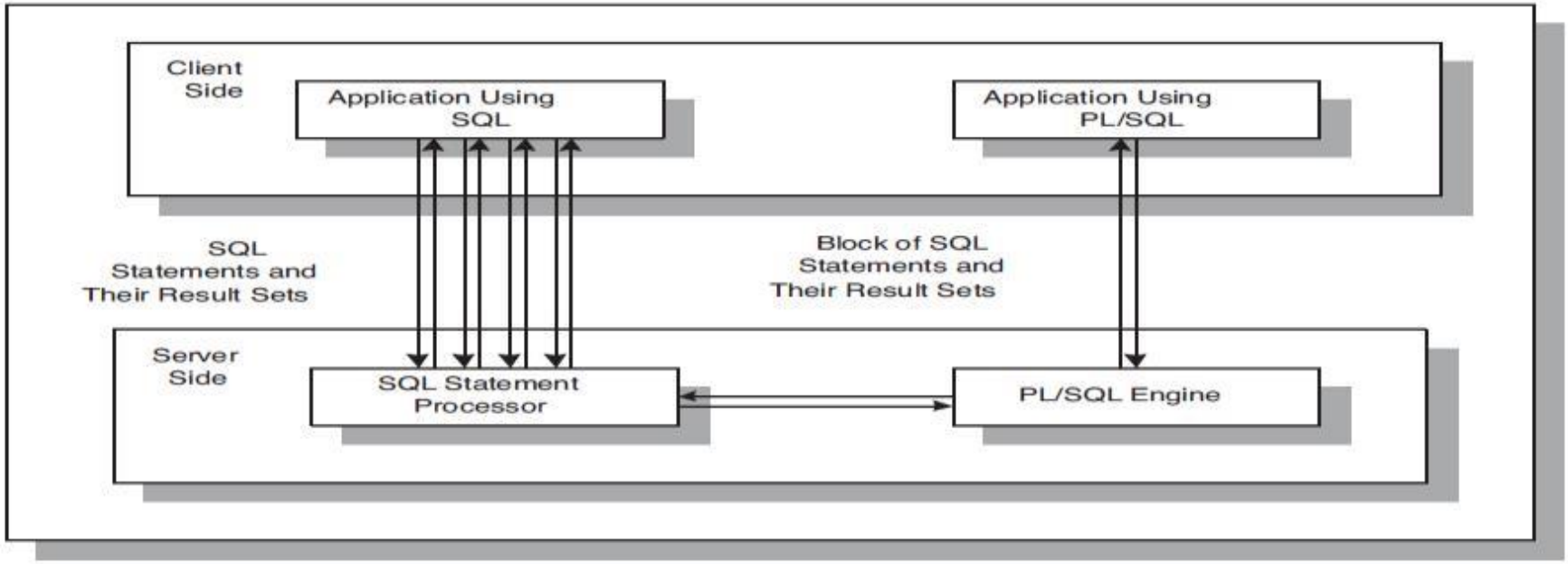
PL/SQL can also directly be called from the command-line SQL*Plus interface. Direct call can also be made from external programming language calls to database.

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.

PL/SQL-Overview

PL/SQL is there any Advantage?



Each SELECT statement is a request against the database and is sent to the Oracle server

.

The results of each SELECT statement are sent back to the client.

Each time a SELECT statement is executed, network traffic is generated.

Hence, multiple SELECT statements result in multiple round-trip transmissions, adding significantly to the network traffic.

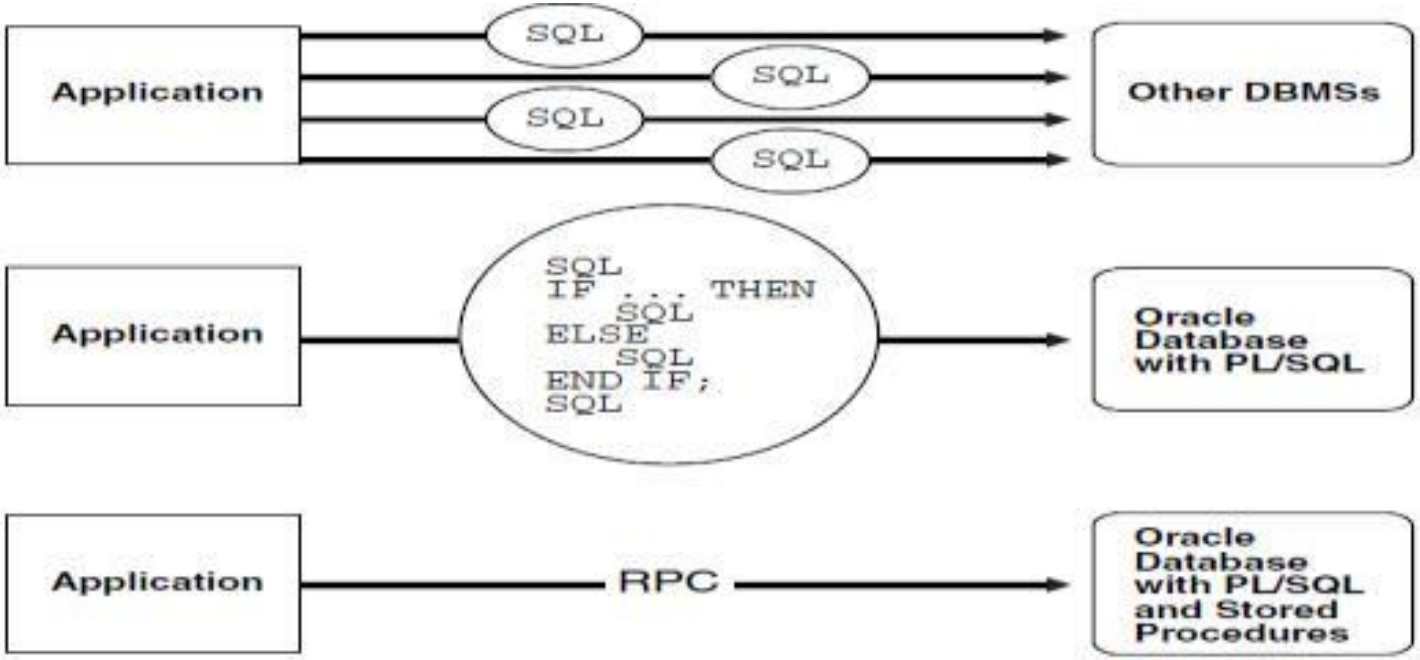
When these SELECT statements are combined into a PL/SQL program, they are sent to the server as a single unit.

The SELECT statements in this PL/SQL program are executed at the server.

The server sends the results of these SELECT statements back to the client, also as a single unit.

Therefore, a PL/SQL program encompassing multiple SELECT statements can be executed at the server and have the results returned to the client in one round trip.

This is a more efficient process than having each SELECT statement executed independently.


Application
SQL
SQL
SQL
SQL
Other DBMSs
Application
SQL
IF ... THEN
SQL
ELSE
SQL
END IF;
SQL
Oracle Database with PL/SQL
Application
RPC
Oracle Database with PL/SQL and Stored Procedures

There are a number of places where we can write code that our applications can use:

- As part of applications
- PL/SQL code embedded in views
- Batch routines

As part of applications

- PL/SQL program units can return a set of values (functions), or PL/SQL routines can perform database operations (procedures).
- These functions and procedures may be called by other functions and procedures.

PL/SQL code embedded in views

- Oracle allows us to embed code in database views.
- We can also embed PL/SQL in INSTEAD OF triggers on a view and allow us to perform INSERT, UPDATE, and DELETE operations on complex views, with PL/SQL programmatically handling how these operations should be handled.

Batch routines

- Batch routines run code that processes a large number of records at the same time.
- These routines are usually large, complex, and database intensive. This type of routine should assuredly be written in PL/SQL.
- Examples
  - Generating invoices for every customer in a system
  - Processing payroll for an entire organization

Features of PL/SQL

It is tightly integrated with SQL. It provides access to predefined SQL packages.
It provides support for developing Web Applications and Server Pages.

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.
PL/SQL supports both static and dynamic SQL.
Static SQL supports DML operations and transaction control from PL/SQL block.
Dynamic SQL is SQL allows embedding DDL statements in PL/SQL blocks

How to Start with PL/SQL Programming???

Save, Edit and Execute PL/SQL program

Type your program in SQL *plus

To save : **save <FileName>**

Program is saved in the home folder, to save in other folder give path.

save \test\firstprg.sql

To make changes: **edit <FileName>**

To edit program saved in folder other then bin

edit \test\firstprg.sql

To Execute: @ **<FileName>**

To execute program saved in folder other then bin.

\test\firstprg.sql

PL/SQL block structure

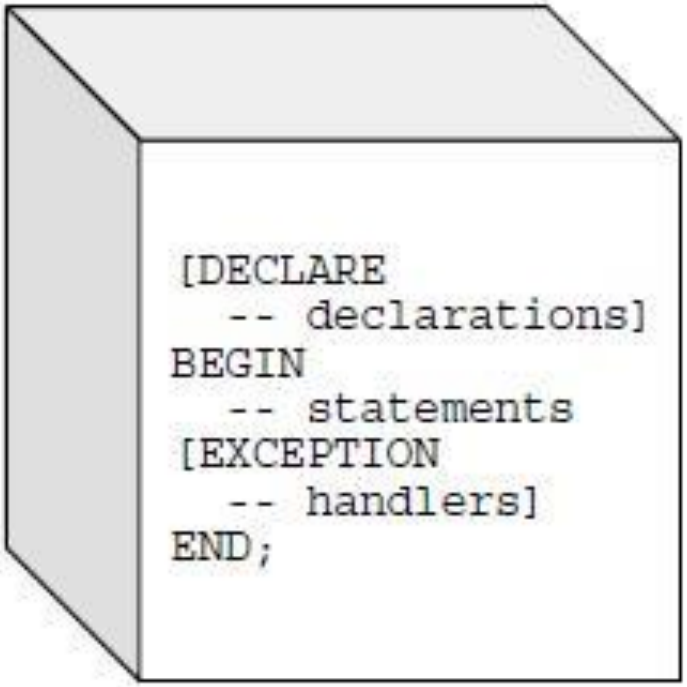
PL/SQL blocks can be divided into two groups:

- Anonymous
- Named

Named PL/SQL blocks are used when creating subroutines like procedures, functions, and packages. The subroutines then can be stored in the database and referenced by their names later.

In addition, subroutines such as procedures and functions can be defined within the anonymous PL/SQL block.

Block Structure



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.

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.

Exception Handling

This section starts with the keyword EXCEPTION.

This section is also optional and contains exceptions that handle errors in the program.

Declare - example

```
declare
    v_sal_nr NUMBER;
    v_name_tx VARCHAR2(10) DEFAULT ‘KING’;
    v_height_nr NUMBER := 4;
    v_start_dt DATE := SYSDATE; -- same as DEFAULT
begin
...
```

Declare - example

```
declare
    variable_name table.column%TYPE;
    variable_name2 variable_name%TYPE;
    variable_row table%ROWTYPE;
begin
```

To see the output on sqlplus execute following commands

```
set serveroutput on
set serveroutput on size 5000;
```

- Example-1
  - begin
    - dbms_ouput.put_line(“hello”);
    - end
    -
    - Example 2
      - Declare
      - Message varchar2(20):=”Hello, World”;
      - Begin
      - dbms_output.put_line(message);
      - End;

PL/SQL code blocks are comprised of statements.

Each statement ends with a semi-colon.

PL/SQL code blocks are followed by a slash (/) in the first position of the following line. This causes the code block statements to be executed.

The only PL/SQL code block keyword that is followed by a semi-colon is the End keyword.

Fundamentals of PL/SQL

- Full-featured programming language
- An interpreted language
- Type in editor, execute in SQL*Plus

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

Variables and Data Types

- Variables
  - Used to store numbers, character strings, dates, and other data values
  - Avoid using keywords, table names and column names as variable names
  - Must be declared with data type before use: *variable_name data_type_declaration*;

Scalar Data Types

- Represent a single value

Data Type	Description	Sample Declaration
VARCHAR2	Variable-length character string	`current_s_last VARCHAR2(30);`
CHAR	Fixed-length character string	`student_gender CHAR(1);`
DATE	Date and time	`todays_date DATE;`
INTERVAL	Time interval	`curr_time_enrolled INTERVAL YEAR TO MONTH;` `curr_elapsed_time INTERVAL DAY TO SECOND;`
NUMBER	Floating-point, fixed-point, or integer number	`current_price NUMBER(5,2);`

Table 4-2 Scalar database data types

Scalar Data Types

Data Type	Description	Sample Declaration
Integer number subtypes (BINARY_INTEGER, INTEGER, INT, SMALLINT)	Integer	`counter BINARY_INTEGER;`
Decimal number subtypes (DEC, DECIMAL, DOUBLE PRECISION, NUMERIC, REAL)	Numeric value with varying precision and scale	`student_gpa REAL;`
BOOLEAN	True/False value	`order_flag BOOLEAN;`

Table 4-3 General scalar data types

Composite and Reference Variables

- Composite variables
  - RECORD: contains multiple scalar values, similar to a table record
  - TABLE: tabular structure with multiple columns and rows
  - VARRAY: variable-sized array
- Reference variables
  - Directly reference a specific database field or record and assume the data type of the associated field or record
  - %TYPE: same data type as a database field
  - %ROWTYPE: same data type as a database record

PL/SQL Program Blocks

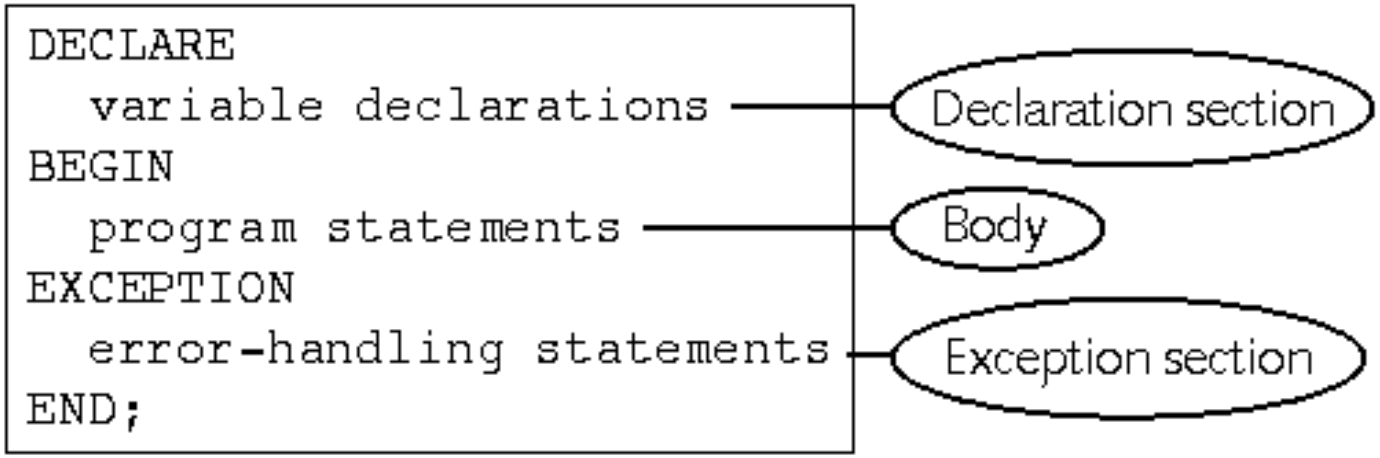


Figure 4-1 Structure of a PL/SQL program block

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

Arithmetic Operators

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

Assignment Statements

- Assigns a value to a variable
- *variable_name* := *value*;
- Value can be a literal:
  - current_s_first_name := 'John';
- Value can be another variable:
  - current_s_first_name := s_first_name;

Executing a PL/SQL Program in SQL*Plus

```
--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 4-2 PL/SQL program commands

- Create program in text editor
- Paste into SQL*Plus window
- Press Enter, type / then enter to execute

PL/SQL 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

Manipulating Character Strings with PL/SQL

- To concatenate two strings in PL/SQL, you use the double bar (||) operator:
  - *new_string* := *string1* || *string2*;
- To remove blank leading spaces use the LTRIM function:
  - *string := LTRIM(string_variable_name);*
- To remove blank trailing spaces use the RTRIM function:
  - *string := RTRIM(string_variable_name);*
- To find the number of characters in a character string use the LENGTH function:
  - *string_length := LENGTH(string_variable_name);*

Manipulating Character Strings with PL/SQL

- To change case, use UPPER, LOWER, INITCAP
- INSTR function searches a string for a specific substring:
  - *start_position := INSTR(original_string, substring);*
- SUBSTR function extracts a specific number of characters from a character string, starting at a given point:
  - *extracted_string := SUBSTR(string_variable, starting_point, number_of_characters);*

Debugging PL/SQL Programs

- Syntax error:
  - Command does not follow the guidelines of the programming language
  - Generates compiler or interpreter error messages
- Logic error:
  - Program runs but results in an incorrect result
  - Caused by mistake in program

Finding and Fixing Syntax Errors

- Interpreter flags the line number and character location of syntax errors
- If error message appears and the flagged line appears correct, the error usually occurs on program lines *preceding* the flagged line
- Comment out program lines to look for hidden errors
- One error (such as missing semicolon) may cause more – fix one error at a time

Finding and Fixing Logic Errors

- Locate logic errors by viewing variable values during program execution
- There is no SQL*Plus debugger
- Use DBMS_OUTPUT statements to print variable values

Lesson B

- Create PL/SQL decision control structures
- Use SQL queries in PL/SQL programs
- Create loops in PL/SQL programs
- Create PL/SQL tables and tables of records
- Use cursors to retrieve database data into PL/SQL programs
- Use the exception section to handle errors in PL/SQL programs

PL/SQL Decision Control Structures

- Use IF/THEN structure to execute code if condition is true
  - IF *condition* THEN

    commands that execute if condition is TRUE;

    END IF;
- If condition evaluates to NULL it is considered false
- Use IF/THEN/ELSE to execute code if condition is true or false
  - IF *condition* THEN

    commands that execute if condition is TRUE;

    ELSE

    commands that execute if condition is FALSE;

    END IF;
- Can be nested – be sure to end nested statements

PL/SQL Decision Control Structures

- Use IF/ELSIF to evaluate many conditions:
  - IF *condition1* THEN

    commands that execute if condition1 is TRUE;

    ELSIF *condition2* THEN

    commands that execute if condition2 is TRUE;

    ELSIF *condition3* THEN

    commands that execute if condition3 is TRUE;

    ...

    ELSE

    commands that execute if none of the

    conditions are TRUE;

    END IF;

IF/ELSIF Example

Oracle SQL*Plus

File Edit Search Options Help

```
SQL> DECLARE
  2     todays_date DATE;
  3     current_day VARCHAR2(9);
  4  BEGIN
  5     todays_date := SYSDATE;
  6     -- extract day portion from current date, and trim trailing blank spaces
  7     current_day := TO_CHAR(todays_date, 'DAY');
  8     current_day := INITCAP(current_day);
  9     current_day := RTRIM(current_day);
 10     -- IF/ELSIF condition to determine current day
 11     IF current_day = 'Friday' THEN
 12       DBMS_OUTPUT.PUT_LINE('Today is Friday!');
 13     ELSIF current_day = 'Saturday' THEN
 14       DBMS_OUTPUT.PUT_LINE('Today is Saturday!');
 15     ELSIF current_day = 'Sunday' THEN
 16       DBMS_OUTPUT.PUT_LINE('Today is Sunday!');
 17     ELSIF current_day = 'Monday' THEN
 18       DBMS_OUTPUT.PUT_LINE('Today is Monday!');
 19     ELSIF current_day = 'Tuesday' THEN
 20       DBMS_OUTPUT.PUT_LINE('Today is Tuesday!');
 21     ELSIF current_day = 'Wednesday' THEN
 22       DBMS_OUTPUT.PUT_LINE('Today is Wednesday!');
 23     ELSIF current_day = 'Thursday' THEN
 24       DBMS_OUTPUT.PUT_LINE('Today is Thursday!');
 25     ELSE
 26       DBMS_OUTPUT.PUT_LINE('Current day not found.');
 27     END IF;
 28  END;
 29  /
Today is Tuesday!

PL/SQL procedure successfully completed.
```

Add/modify these commands

Figure 4-17 Using an IF/ELSIF structure

Complex Conditions

- Created with logical operators AND, OR and NOT
- AND is evaluated before OR
- Use () to set precedence

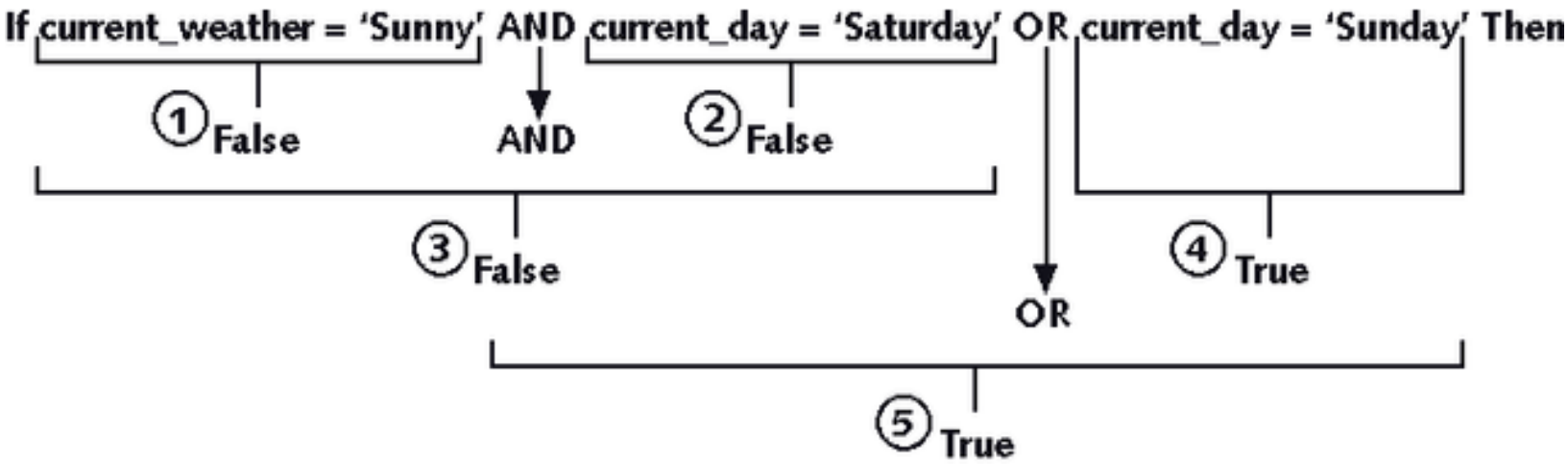


Figure 4-19 Evaluating AND and OR in an expression

Using SQL Queries in PL/SQL Programs

- Action queries can be used as in SQL*Plus
- May use variables in action queries
- DDL commands may not be used in PL/SQL

Loops

- Program structure that executes a series of program statements, and periodically evaluates an exit condition to determine if the loop should repeat or exit
- Pretest loop: evaluates the exit condition before any program commands execute
- Posttest loop: executes one or more program commands before the loop evaluates the exit condition for the first time
- PL/SQL has 5 loop structures

The LOOP...EXIT Loop

```
LOOP
  [program statements]
  IF condition THEN
    EXIT;
  END IF;
  [additional program statements]
END LOOP
```

The LOOP...EXIT WHEN Loop

```
LOOP
    program statements
    EXIT WHEN condition;
END LOOP;
```

The WHILE...LOOP

WHILE *condition* LOOP

program statements

END LOOP;

The Numeric FOR Loop

FOR *counter_variable* IN *start_value* .. *end_value*
LOOP
    program statements
END LOOP;

Stored Procedures and Functions Procedures

The procedures and functions that were part of an anonymous block and were called within the executable section.

It is also possible to store the procedure or function definition in the database and have it invoked from various environments that have access to the database.

Stored Procedures and Functions Procedures

A Procedure or function is a schema object that logically groups a set of SQL and other PL/SQL programming language statements together to perform a specific task.

Procedures and Functions are created in user’s schema and stored in a database for continued use.

The general syntax for creating procedure is

```
Create [or replace] procedure <proc-name>
[(<parameter-list>)] as
  <declarations>
  Begin
  <executable-section>
[exception
   <exception-section>]
End;
```

Create a table called employee having following columns

EMPCODE	NUMBER(5)
EMPNAME	VARCHAR2(20)
DOB	DATE
DEPT	VARCHAR2(20)
SALARY	NUMBER(6)

Insert 5 or 6 records in the table.

Declare a procedure to increase the salary of an employee by 2000 by specifying the employee code.

```
create or replace procedure sal_hike  (eno in
employee1.empcode%type, increase in employee1.salary%type)
as  sal employee1.salary%type;
new_sal employee1.salary%type;
begin
select salary into sal from employee1 where empcode=eno;
new_sal:=sal+increase;
update employee1 set salary=new_sal where empcode=eno;
dbms_output.put_line('Table updated with new salary' ||new_sal);
end;
/
```

The PL/SQL code can be stored in a file with extension .sql and can be executed by the command

@ filename.sql

exec proc-name(parameters);

Functions

The general syntax for creating function is

```
Create [or replace] function <func-name>
[(<parameter-list>)] return <datatype> as
<declarations>
Begin
   <executable-section>
[exception
    <exception-section>
End;
```

When stored function is invoked from within an SQL statement there are three restrictions:
There should not be no ‘out’ parameters.
The function should be applicable to a row in the table.
Return data type should be compatible with an SQL data type.

Create a table called STUDENT having the following columns

REGNO NOT NULL	VARCHAR2(10)
NAME	VARCHAR2(20)
MAJOR	VARCHAR2(20)
BDATE	DATE

Insert 5 or 6 records in the table.

Write a function to get the major subject offered by a student with a specific register number

```
create or replace function get_major(rno in
student1.regno%type)
return student1.major%type as smajor
student1.major%type;
begin
select major into smajor from student1 where
student1.regno=rno;
  return(smajor);
end;
/
```

The function should be written in a .sql file and will be executed as @filename.sql

Records – PL/SQL

Records

A record is a composite data structure composed of one or more elements.

Records are very much like a row of a database table, but each element of the record does not stand on its own.

PL/SQL supports three kinds of records:

table-based

cursor-based

programmer-defined

Table based Records

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

```
DECLARE
        emp_rec employee1%rowtype;
BEGIN
        SELECT * into emp_rec FROM employee1 WHERE
empcode = 111;
        dbms_output.put_line('Employee - code ' ||
emp_rec.empcode);
        dbms_output.put_line('Employee - name ' ||
emp_rec.empname);

        dbms_output.put_line('Employee - DOB ' ||
emp_rec.dob);
        dbms_output.put_line('Employee - salary ' ||
emp_rec.salary);
END;
/
```

User-Defined Records

```
DECLARE
        type books is record (title varchar(50),author varchar(50),book_id
number);
        book1 books;
        book2 books;
BEGIN
        book1.title := 'DBMS'; book1.author := 'ABCD’; book1.book_id :=
123;
        book2.title := 'NoSQL'; book2.author := 'PAL'; book2.book_id :=
456;
--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 book_id : ' || book1.book_id);
-- Print book 2record
        dbms_output.put_line('Book 2 title : '|| book2.title);
        dbms_output.put_line('Book 2 author : '|| book2.author);
        dbms_output.put_line('Book 2 book_id : '|| book2.book_id);
END;
/
```

Cursors

- Pointer to a memory location that the DBMS uses to process a SQL query
- Use to retrieve and manipulate database data

Implicit Cursor

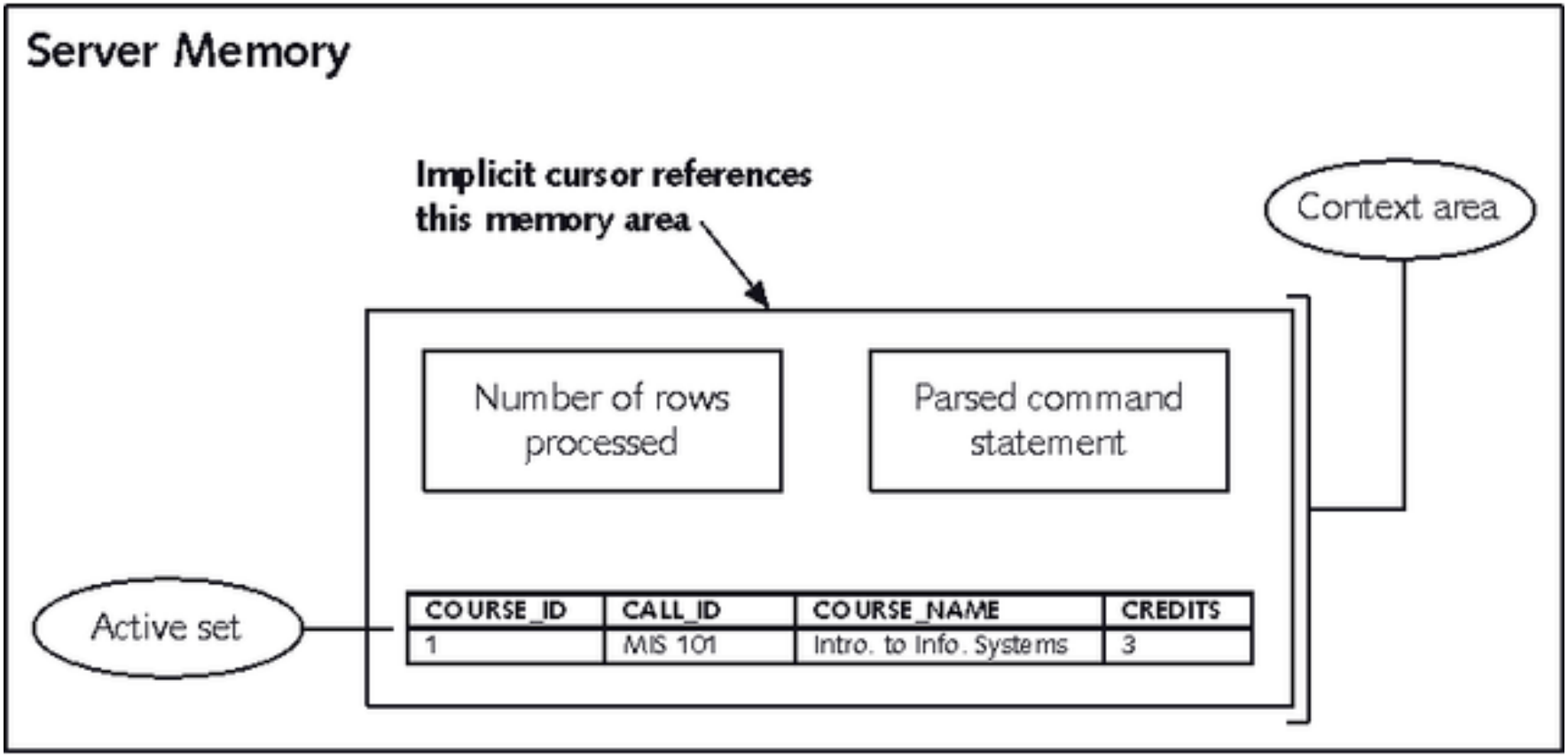


Figure 4-26 Implicit cursor

Using an Implicit Cursor

- Executing a SELECT query creates an implicit cursor
- To retrieve it into a variable use INTO:
  - SELECT *field1*, *field2*, ...

  INTO *variable1*, *variable2*, ...

  FROM *table1*, *table2*, ...

  WHERE *join_ conditions*

  AND *search_condition_to_retrieve_1_record*;
- Can only be used with queries that return exactly one record

Implicit cursors are created in PL/SQL by any DML statement that do not have explicit cursor defined within the statement.

During the processing of an implicit cursor, Oracle automatically performs the Open, Fetch and Close operations for the cursor.

Implicit Cursors are created for the following statements of SQL:

Select…into, Insert, Update and Delete.

Example: Implicit cursor (Update Command)

```
Declare
total_rows number(2);
    Begin
      update employee1 set salary=50000+5000 where empcode=111;
    If sql%isopen then
        dbms_output.put_line('cursor is open');
    else
        dbms_output.put_line('cursor is not open');
    End if;
    If sql%notfound then
        dbms_output.put_line('No Employees selected');
    Elsif sql%found then
         total_rows:=sql%rowcount;
        dbms_output.put_line('total_rows || 'Employees Selected');
    End if;
    End;
    /
```

Implicit cursors uses the following constructs to check whether the SQL query/statement has been found or executed.

1. **%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

2. **%notfound:** The logical opposite of %found.

3. **%isopen:** always returns false for implicit cursors, because oracle closes the SQL cursor automatically after executing its associated SQL statement.

4**.%Rowcount:** returns the number of rows affected by an Insert, update, or delete statement or returned by Select..into statement.

Explicit Cursor

- Use for queries that return multiple records or no records
- Must be explicitly declared and used
- An explicit cursor can be generated in the Declare section of the PL/SQL block.
- The advantage of explicit cursor over implicit cursor is, explicit cursor gives more programmatic control to programmer.
- Implicit cursors are less efficient than explicit cursors.

Using an Explicit Cursor

- Declare the cursor
  - *CURSOR cursor_name IS select_query;*
- Open the cursor
  - *OPEN cursor_name;*
- Fetch the data rows
  - *LOOP*

    FETCH cursor_name INTO variable_name(s);

    EXIT WHEN cursor_name%NOTFOUND;
- Close the cursor
  - *CLOSE cursor_name;*

Explicit Cursor – Fetching the rows

The rows from a table can be fetched into explicit cursor as

fetch c_customers into c_id, c_name, c_addr;

where c_customers is table name

c_id, c_name, c_addr are variables created in PL/SQL, like column types of c_customer table.

The declaration of cursor allows the application to sequentially process each row of data as the cursor returns it.

An explicit cursor can be generated in the Declare section of the PL/SQL block.

Explicit Cursor with %ROWTYPE

Oracle SQL*Plus

File Edit Search Options Help

```
SQL> DECLARE
  2    current_bldg_code VARCHAR2(5);
  3    CURSOR location_cursor IS
  4      SELECT room, capacity
  5      FROM location
  6      WHERE bldg_code = current_bldg_code;
  7    location_row location_cursor%ROWTYPE;
  8  BEGIN
  9    current_bldg_code := 'LIB';
 10    OPEN location_cursor;
 11    LOOP
 12      FETCH location_cursor INTO location_row;
 13      EXIT WHEN location_cursor%NOTFOUND;
 14      DBMS_OUTPUT.PUT_LINE('The capacity of ' || current_bldg_code || ' ' ||
 15      location_row.room || ' is ' || location_row.capacity || ' seat(s).');
 16    END LOOP;
 17    CLOSE location_cursor;
 18  END;
 19  /
The capacity of LIB 217 is 2 seat(s).
The capacity of LIB 222 is 1 seat(s).

PL/SQL procedure successfully completed.
```

Modify these commands

Program output

Figure 4-31 Processing an explicit cursor using a %ROWTYPE variable

Cursor FOR Loop

- Automatically opens the cursor, fetches the records, then closes the cursor
- *FOR variable_name(s) IN cursor_name LOOP*

  processing commands

  END LOOP;
- Cursor variables cannot be used outside loop

Using Cursor FOR Loop

Oracle SQL*Plus

File Edit Search Options Help

```
SQL> DECLARE
  2     current_bldg_code VARCHAR2(5);
  3     CURSOR location_cursor IS
  4       SELECT room, capacity
  5       FROM location
  6       WHERE bldg_code = current_bldg_code;
  7     location_row location_cursor%ROWTYPE;
  8   BEGIN
  9     current_bldg_code := 'LIB';
 10     FOR location_row IN location_cursor LOOP
 11       DBMS_OUTPUT.PUT_LINE('The capacity of ' || current_bldg_code || ' ' ||
 12       location_row.room || ' is ' || location_row.capacity || ' seat(s).');
 13     END LOOP;
 14   END;
 15   /
The capacity of LIB 217 is 2 seat(s).
The capacity of LIB 222 is 1 seat(s).

PL/SQL procedure successfully completed.
```

Add/modify these commands

Figure 4-32 Processing an explicit cursor using a cursor FOR loop

Handling Runtime Errors in PL/SQL Programs

- Runtime errors cause exceptions
- Exception handlers exist to deal with different error situations
- Exceptions cause program control to fall to exception section where exception is handled

```
EXCEPTION
  WHEN exception1_name THEN
     exception1 handler commands;
  WHEN exception2_name THEN
     exception2 handler commands;
  …
  WHEN OTHERS THEN
     other handler commands;
END;
```

Figure 4-34 Exception handler syntax

Exception Handling

An error condition during a program execution is called an exception in PL/SQL.

PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition.

There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

```
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 others THEN
exception3-handling-statements
END;
/
```

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;
/
```

Predefined Exceptions

Oracle Error Code	Exception Name	Description
ORA-00001	DUP_VAL_ON_INDEX	Command violates primary key unique constraint
ORA-01403	NO_DATA_FOUND	Query retrieves no records
ORA-01422	TOO_MANY_ROWS	Query returns more rows than anticipated
ORA-01476	ZERO_DIVIDE	Division by zero
ORA-01722	INVALID_NUMBER	Invalid number conversion (such as trying to convert "2B" to a number)
ORA-06502	VALUE_ERROR	Error in truncation, arithmetic, or data conversion operation

Table 4-10 Common PL/SQL predefined exceptions

Undefined Exceptions

- Less common errors
- Do not have predefined names
- Must declare your own name for the exception code in the declaration section
  - *DECLARE*

    e_exception_name EXCEPTION;

    PRAGMA EXCEPTION_INIT(e_exception_name,

    -Oracle_error_code);

User-Defined Exceptions

- Not a real Oracle error
- Use to enforce business rules

```
DECLARE
   e_exception_name EXCEPTION;
   other variable declarations;
BEGIN
   other program commands
   IF exception_condition THEN
      RAISE e_exception_name;
   END IF;
   other program commands
EXCEPTION
   WHEN e_exception_name THEN
      exception handler commands;
END;
```

Defining the exception

Raising the exception

Handling the exception

Figure 4-40 General syntax for declaring, raising, and handling a user-defined exception

Summary

- PL/SQL is a programming language for working with an Oracle database
- Scalar, composite and reference variables can be used
- The IF/THEN/ELSE decision control structure allows branching logic
- Five loop constructs allow repeating code
- Cursors are returned from queries and can be explicitly iterated over
- Exception handling is performed in the exception section. User defined exceptions help to enforce business logic