

LEARNING PL/SQL

A BEGINNER'S LEARNING GUIDE
RANDALL FADLER | AUGUST 2024



Preface

Welcome to “Introduction to PL/SQL Programming.” This booklet is designed to provide you with a better understanding of Oracle’s Procedural Language/Structured Query Language (PL/SQL), a powerful extension of SQL that allows for procedural programming within the Oracle database environment. Whether you are a database administrator, developer, or someone looking to enhance your skills in database programming, this guide will serve as a valuable resource.

This booklet assumes that you have some prior experience with computers, including installing software and using text editors. It is also expected that you have a general background in programming techniques. Familiarity with basic concepts such as variables, control structures, and error handling will be beneficial as you delve into the intricacies of PL/SQL. While this guide is tailored for those new to PL/SQL, it builds on your existing knowledge of programming and database management, allowing you to quickly grasp and apply the concepts presented.

Throughout this booklet, you will find detailed explanations, practical examples, and hands-on exercises designed to reinforce your learning. From understanding the fundamental block structure of PL/SQL to mastering advanced topics such as cursors, triggers, and packages, each chapter is crafted to build your proficiency step by step. By the end of this guide, you will be equipped with the skills and knowledge to write efficient, maintainable, and robust PL/SQL code, enabling you to leverage the full potential of Oracle databases in your projects.

We hope you find this booklet both informative and engaging, and that it serves as a solid foundation for your journey into the world of PL/SQL programming. Happy coding!

Beginner's Guide to Oracle PL/SQL Programming

I. Introduction

- **What is PL/SQL?**

PL/SQL, or Procedural Language/Structured Query Language, is Oracle Corporation's extension for SQL, designed to enhance the capabilities of SQL by adding procedural constructs. This language allows developers to write complex scripts that can perform a variety of tasks within the Oracle database environment.

PL/SQL is structured into blocks, which are the basic units of code. Each block can contain declarations, executable commands, and exception handling sections. This block structure makes the code modular and easier to manage. Within these blocks, you can declare variables, constants, and complex data types, which can be used to store and manipulate data.

One of the key features of PL/SQL is its ability to create reusable subprograms, such as procedures and functions. Procedures are used to perform actions, while functions return values. These subprograms help encapsulate business logic, making the code more organized and reusable. Additionally, PL/SQL supports packages, which are collections of related procedures, functions, variables, and other PL/SQL constructs. Packages provide a way to group logically related items together, promoting better modularity and reusability.

PL/SQL also offers robust error handling through the use of exceptions. Developers can define exception handlers to manage different types of errors that may occur during the execution of the code. This feature ensures that the application can handle unexpected situations gracefully, maintaining the integrity of the data and the stability of the system.

Control structures in PL/SQL, such as conditional statements (IF-THEN-ELSE) and loops (FOR, WHILE), allow developers to write complex logic to control the flow of the program. These structures enable the execution of different code paths based on certain conditions, making the language powerful and flexible.

Cursors in PL/SQL are used to handle query results row by row. They provide a mechanism to fetch and process individual rows returned by a query, which is particularly useful for operations that require row-by-row processing, such as data validation and complex calculations.

Overall, PL/SQL is a powerful and versatile language that combines the data manipulation capabilities of SQL with the procedural features of a programming language. It is widely used for writing data-centric programs that perform operations such as data validation, complex calculations, and batch processing within the Oracle database environment. PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension for SQL and the Oracle relational database. It combines the data manipulation power of SQL with the processing power of procedural languages.

- Differences between SQL and PL/SQL:

SQL (Structured Query Language) and PL/SQL (Procedural Language/Structured Query Language) are both used in the context of Oracle databases, but they serve different purposes and have distinct characteristics.

SQL is a standard language used to interact with relational databases. It is primarily a declarative language, meaning it focuses on what needs to be done rather than how it should be done. SQL is used for tasks such as querying data, updating records, and managing database structures. It includes commands like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. SQL operates on sets of data and is designed to handle single operations at a time. It does not support procedural constructs like loops or conditional statements, and it lacks error handling capabilities.

PL/SQL, on the other hand, is Oracle's procedural extension to SQL. It combines the data manipulation power of SQL with the procedural capabilities of a programming language. PL/SQL allows you to write code in blocks, which can include declarations, executable commands, and exception handling. This block structure makes it possible to create complex scripts that can perform multiple operations in a single execution, reducing network traffic and improving performance.

PL/SQL supports variables, constants, and complex data types, enabling you to store and manipulate data within your programs. It also includes control structures such as loops (`FOR`, `WHILE`) and conditional statements (`IF-THEN-ELSE`), which allow for more complex logic and flow control. Additionally, PL/SQL provides robust error handling through exceptions, enabling you to manage and respond to runtime errors effectively.

- **Why Learn PL/SQL?**

Learning PL/SQL can be highly beneficial for several reasons, especially if you work with Oracle databases or plan to pursue a career in database management and development. Here are some compelling reasons to learn PL/SQL:

Enhanced SQL Capabilities: PL/SQL extends the capabilities of SQL by adding procedural constructs such as loops, conditional statements, and error handling. This allows you to write more complex and efficient database programs.

Performance Improvement: By using PL/SQL, you can reduce the number of calls between your application and the database. This minimizes network traffic and can significantly improve the performance of your applications.

Modularity and Reusability: PL/SQL supports the creation of procedures, functions, and packages, which can encapsulate business logic and be reused across multiple applications. This promotes code modularity and reduces redundancy.

Robust Error Handling: PL/SQL provides robust error handling through exceptions, allowing you to manage and respond to runtime errors effectively. This ensures that your applications can handle unexpected situations gracefully.

Integration with Oracle Tools: PL/SQL is tightly integrated with Oracle tools and technologies, such as Oracle Forms, Oracle Reports, and Oracle Application Express (APEX). This makes it easier to develop comprehensive database applications.

Career Opportunities: Proficiency in PL/SQL is highly valued in the job market, particularly for roles such as database developer, database administrator, and application developer. Many organizations rely on Oracle databases, and having PL/SQL skills can open up numerous career opportunities.

Data Security: PL/SQL allows you to implement complex security measures within the database, ensuring that sensitive data is protected, and access is controlled.

Batch Processing: PL/SQL is well-suited for batch processing tasks, such as data migration, data cleansing, and report generation. It can handle large volumes of data efficiently.

Community and Resources: There is a large and active community of PL/SQL developers, along with extensive documentation and resources available online. This makes it easier to learn and get support when needed.

[Overview of PL/SQL \(oracle.com\)](https://www.oracle.com/plsql/overview/)

2. Setting Up Your Environment

- **Installing Oracle Database**

- Step-by-step guide:

1. Download the Oracle Database installer from the Oracle website:
 - <https://www.oracle.com/database/free/get-started/>
2. Run the installer and follow the on-screen instructions.
3. Configure the database settings as needed.
4. Complete the installation and verify that the database is running.

- **Setting Up Oracle SQL Developer**

- Installation and configuration:

1. Download Oracle SQL Developer from the Oracle website.
2. Extract the downloaded file and run the SQL Developer executable.
3. Configure the connection to your Oracle Database by providing the necessary details (hostname, port, SID/service name, username, and password).

PL/SQL Basic Concepts

The PL/SQL block structure is fundamental to writing PL/SQL code. It organizes code into logical units, making it easier to read, maintain, and debug. Each PL/SQL block consists of three main sections: the declaration section, the executable section, and the exception handling section. Here's a detailed explanation of each:

Declaration Section:

This is the optional part of the block where you declare variables, constants, cursors, and other PL/SQL constructs.

Declarations are made at the beginning of the block and are used to define the data types and initial values for variables.

Example:

```
DECLARE

    v_employee_id NUMBER;

    v_employee_name VARCHAR2(50);
```

Executable Section:

This is the mandatory part of the block where you write the actual code to be executed.

It contains SQL statements, procedural statements, control structures (like loops and conditional statements), and calls to subprograms (procedures and functions).

This section is enclosed between the BEGIN and END keywords.

Example:

```
BEGIN

    SELECT employee_id, employee_name
    INTO v_employee_id, v_employee_name
    FROM employees
    WHERE employee_id = 100;
```

Exception Handling Section:

This is the optional part of the block where you handle exceptions (errors) that occur during the execution of the executable section.

It allows you to define specific actions to take when certain exceptions are raised, ensuring that the program can handle errors gracefully.

This section starts with the EXCEPTION keyword.

Example:

```
EXCEPTION

    WHEN NO_DATA_FOUND THEN

        DBMS_OUTPUT.PUT_LINE('No employee found with the given ID.');
```

```
    WHEN OTHERS THEN

        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
```

Here's a complete example of a PL/SQL block that includes all three sections:

```
DECLARE

    v_employee_id NUMBER;

    v_employee_name VARCHAR2(50);

BEGIN

    SELECT employee_id, employee_name

    INTO v_employee_id, v_employee_name

    FROM employees

    WHERE employee_id = 100;

    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_employee_name);

EXCEPTION

    WHEN NO_DATA_FOUND THEN

        DBMS_OUTPUT.PUT_LINE('No employee found with the given ID.');
```

```
    WHEN OTHERS THEN

        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
```

```
END;
```

In this example:

The declaration section declares two variables, `v_employee_id` and `v_employee_name`.

The executable section retrieves the employee's ID and name from the `employees` table and prints the name.

The exception handling section handles the `NO_DATA_FOUND` exception and any other unexpected errors.

[PL/SQL Language Fundamentals \(oracle.com\)](https://www.oracle.com/plsql/language-fundamentals/)

Function, Procedures, and Packages

Oracle PL/SQL functions are subprograms that return a single value and can be used in SQL statements wherever an expression is allowed. Functions are designed to perform specific tasks and return a result, making them useful for encapsulating reusable logic that can be invoked from various parts of an application. They can accept parameters, which allows them to process input values and return a computed result. Functions are often used for calculations, data transformations, and validations. For example, a function might calculate the total sales for a given period or validate a user's input. Functions must include a `RETURN` statement that specifies the value to be returned to the calling environment.

Procedures in PL/SQL are like functions but are designed to perform actions rather than return values. While functions are used to compute and return a result, procedures are used to execute a series of statements that may include data manipulation, control structures, and calls to other subprograms. Procedures can accept input parameters, output parameters, or both, allowing them to interact with the calling environment. They are often used for tasks such as updating records, processing transactions, and implementing business logic. Unlike functions, procedures do not need to return a value, but they can return multiple values through output parameters. This makes procedures highly versatile for a wide range of operations within the database.

Packages in PL/SQL are collections of related procedures, functions, variables, and other PL/SQL constructs grouped together as a single unit. A package consists of two parts: the specification and the body. The specification declares the public elements of the package, such as procedures, functions, and variables, which can be accessed by other PL/SQL code. The body contains the implementation of these elements, including the actual code for procedures and functions. Packages provide several benefits, including modularity, encapsulation, and reusability. By grouping related subprograms and data together, packages help organize code logically and make it easier to manage and maintain. They also allow for better performance, as the entire package is loaded into memory when a component is called, reducing the overhead of repeated parsing and execution.

Data Types

Oracle PL/SQL offers a rich variety of data types to handle different kinds of data efficiently and effectively. These data types are broadly categorized into scalar and composite types. Scalar data types store single values and include numeric types like `NUMBER`, `BINARY_FLOAT`, and `BINARY_DOUBLE`, which are used for real numbers, integers, and floating-point numbers, respectively. The `PLS_INTEGER` type, specific to PL/SQL, is optimized for performance and is used for signed 32-bit integers. Character data types such as `CHAR`, `VARCHAR2`, `NCHAR`, and `NVARCHAR2` handle fixed and variable-length strings, while the `BOOLEAN` type represents logical values `TRUE`, `FALSE`, and `NULL`. Date and time data types like `DATE`, `TIMESTAMP`, and `INTERVAL` are used to store date and time information with varying levels of precision. Composite data types, on the other hand, can store multiple values and include collections like `VARRAY` and `Nested Table`, as well as records that allow you to group related data items together. Additionally, PL/SQL supports large object (LOB) data types such as `BLOB`, `CLOB`, `NCLOB`, and `BFILE` for storing large amounts of binary and character data.

EXAMPLES OF NUMERIC DATA TYPES

```
DECLARE

    v_number NUMBER(10, 2); -- A number with up to 10 digits, 2 of which can be after
    the decimal point

    v_integer PLS_INTEGER; -- A signed 32-bit integer

BEGIN

    v_number := 12345.67;

    v_integer := 100;

    DBMS_OUTPUT.PUT_LINE('Number: ' || v_number);

    DBMS_OUTPUT.PUT_LINE('Integer: ' || v_integer);

END;
```

EXAMPLES OF CHARACTER DATA TYPES

```
DECLARE

    v_char CHAR(10); -- A fixed-length character string

    v_varchar2 VARCHAR2(50); -- A variable-length character string

BEGIN

    v_char := 'Hello';

    v_varchar2 := 'Hello, PL/SQL!';

    DBMS_OUTPUT.PUT_LINE('Char: ' || v_char);

    DBMS_OUTPUT.PUT_LINE('Varchar2: ' || v_varchar2);
```

```
END;
```

EXAMPLES OF DATA AND TIME DATA TYPES

```
DECLARE
```

```
    v_date DATE; -- A date value
```

```
    v_timestamp TIMESTAMP; -- A timestamp value
```

```
BEGIN
```

```
    v_date := SYSDATE; -- Current date and time
```

```
    v_timestamp := SYSTIMESTAMP; -- Current timestamp
```

```
    DBMS_OUTPUT.PUT_LINE('Date: ' || TO_CHAR(v_date, 'DD-MON-YYYY HH24:MI:SS'));
```

```
    DBMS_OUTPUT.PUT_LINE('Timestamp: ' || TO_CHAR(v_timestamp, 'DD-MON-YYYY  
HH24:MI:SS.FF'));
```

```
END;
```

EXAMPLE OF BOOLEAN DATA TYPE

```
DECLARE
```

```
    v_boolean BOOLEAN; -- A boolean value
```

```
BEGIN
```

```
    v_boolean := TRUE;
```

```
    IF v_boolean THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Boolean is TRUE');
```

```
    ELSE
```

```
        DBMS_OUTPUT.PUT_LINE('Boolean is FALSE');
```

```
    END IF;
```

```
END;
```

[PL/SQL Data Types \(oracle.com\)](https://www.oracle.com/plsql/PLSQL_Tools_Resources/plsql_data_types.html)

4. Writing Your First PL/SQL Program

Using SQL*Plus to write and execute anonymous PL/SQL blocks is a great way for beginners to get started with PL/SQL. Here's a step-by-step guide:

Step 1: Connect to the Oracle Database

First, you need to connect to your Oracle database using SQL*Plus. Open your command prompt or terminal and type the following command:

```
sqlplus username/password@database
```

Replace username, password, and database with your actual Oracle credentials and database name.

Step 2: Enable Server Output

To see the output of your PL/SQL blocks, you need to enable server output. This can be done by executing the following command:

```
SQL  
  
SET SERVEROUTPUT ON;
```

Step 3: Write an Anonymous PL/SQL Block

An anonymous PL/SQL block is a block of code that is not stored in the database and is used for one-time execution. Here is a simple example of an anonymous PL/SQL block:

```
BEGIN  
  
    DBMS_OUTPUT.PUT_LINE('Hello, PL/SQL!');  
  
END;  
  
/
```

In this block:

The **BEGIN** keyword marks the start of the executable section.

DBMS_OUTPUT.PUT_LINE is a built-in procedure that prints a message to the screen.

The **END;** keyword marks the end of the block.

The **/** character tells SQL*Plus to **execute the block**.

Step 4: Execute the Block

After writing your PL/SQL block, execute it by typing the block into SQL*Plus and then typing / on a new line. For example:

```
SQL> BEGIN
      2  DBMS_OUTPUT.PUT_LINE('Hello, PL/SQL!');
      3  END;
      4  /
```

When you press Enter after the /, SQL*Plus will execute the block and display the output:

Hello, PL/SQL!

Step 5: Handling Errors

If there are any errors in your PL/SQL block, SQL*Plus will display an error message. For example, if you misspell DBMS_OUTPUT, you might see something like this:

```
BEGIN
      DBMS_OUT.PUT_LINE('Hello, PL/SQL!');
END;
/
ORA-06550: line 2, column 3:
PLS-00201: identifier 'DBMS_OUT' must be declared
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
```

This error message indicates that DBMS_OUT is not recognized, and you need to correct it to DBMS_OUTPUT.

Example with Variables and Control Structures

Here's a more complex example that includes variable declarations and a loop:

```
DECLARE
      v_counter NUMBER := 1;
BEGIN
      FOR i IN 1..5 LOOP
            DBMS_OUTPUT.PUT_LINE('Iteration: ' || i);
      END LOOP;
END;
```

/

This block declares a variable `v_counter`, then uses a FOR loop to print the iteration number five times.

5. Control Structures

Oracle PL/SQL control structures are essential for managing the flow of execution in a PL/SQL program. These structures allow you to dictate the order in which statements are executed based on certain conditions or loops. The primary control structures in PL/SQL include conditional statements (IF, CASE), iterative statements (LOOP, FOR, WHILE), and sequential control statements (GOTO, NULL).

Conditional Control Structures

Conditional control structures enable you to execute specific blocks of code based on certain conditions. The IF-THEN statement is the simplest form, where a block of code is executed only if a specified condition is true. For example:

```
DECLARE

    v_salary NUMBER := 5000;

BEGIN

    IF v_salary > 4000 THEN

        DBMS_OUTPUT.PUT_LINE('High salary');

    END IF;

END;

/
```

The IF-THEN-ELSE statement allows you to execute one block of code if the condition is true and another block if it is false:

```
DECLARE

    v_salary NUMBER := 3000;

BEGIN

    IF v_salary > 4000 THEN

        DBMS_OUTPUT.PUT_LINE('High salary');

    ELSE

        DBMS_OUTPUT.PUT_LINE('Low salary');

    END IF;

END;

/
```

The CASE statement is used for more complex conditional logic, where multiple conditions are evaluated:

```
DECLARE

    v_grade CHAR(1) := 'B';

BEGIN

    CASE v_grade

        WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');

        WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Good');

        WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Fair');

        ELSE DBMS_OUTPUT.PUT_LINE('Poor');

    END CASE;

END;

/
```

Iterative Control Structures

Iterative control structures allow you to execute a block of code repeatedly. The LOOP statement is the simplest form, which repeats a block of code indefinitely until an EXIT statement is encountered:

```
DECLARE

    v_counter NUMBER := 1;

BEGIN

    LOOP

        DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);

        v_counter := v_counter + 1;

        EXIT WHEN v_counter > 5;

    END LOOP;

END;

/
```

The FOR loop iterates a specific number of times:

```
BEGIN

    FOR i IN 1..5 LOOP

        DBMS_OUTPUT.PUT_LINE('Iteration: ' || i);

    END LOOP;

END;
```

```

        END LOOP;

    END;

/

```

The WHILE loop continues to execute as long as a specified condition is true:

```

DECLARE

    v_counter NUMBER := 1;

BEGIN

    WHILE v_counter <= 5 LOOP

        DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);

        v_counter := v_counter + 1;

    END LOOP;

END;

/

```

Sequential Control Structures

Sequential control structures manage the flow of execution in a linear fashion. The GOTO statement allows you to jump to a labeled section of code, although its use is generally discouraged due to potential readability issues:

```

DECLARE

    v_counter NUMBER := 1;

BEGIN

    <<start_loop>>

    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);

    v_counter := v_counter + 1;

    IF v_counter <= 5 THEN

        GOTO start_loop;

    END IF;

END;

/

```

The NULL statement is a placeholder that does nothing and can be used to improve code readability:

```

BEGIN

```

```

IF 1 = 2 THEN

    NULL; -- Do nothing

ELSE

    DBMS_OUTPUT.PUT_LINE('This will always execute');

END IF;

END;

/

```

7. Working with Cursors

- **Implicit Cursors**

Implicit cursors in PL/SQL are automatically created by Oracle whenever an SQL statement is executed. These cursors are used to manage the context area, which holds the information needed for processing the SQL statement. Unlike explicit cursors, which you define and control explicitly, implicit cursors are managed entirely by Oracle. They are created for SQL statements such as `SELECT INTO`, `INSERT`, `UPDATE`, and `DELETE`. When you execute one of these statements, Oracle opens an implicit cursor, processes the statement, and then closes the cursor automatically.

One of the key features of implicit cursors is that they simplify the coding process by handling the cursor management behind the scenes. You don't need to declare, open, fetch from, or close an implicit cursor manually. Instead, you can focus on writing the SQL statements and let Oracle take care of the cursor operations. However, you can still access information about the most recently executed implicit cursor using cursor attributes like `SQL%ROWCOUNT`, `SQL%FOUND`, `SQL%NOTFOUND`, and `SQL%ISOPEN`. These attributes provide useful information, such as the number of rows affected by the SQL statement or whether the cursor is still open.

Here's an example of using an implicit cursor with a `SELECT INTO` statement:

```

DECLARE

    v_employee_name VARCHAR2(50);

BEGIN

    SELECT employee_name

    INTO v_employee_name

    FROM employees

    WHERE employee_id = 100;

    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_employee_name);

EXCEPTION

    WHEN NO_DATA_FOUND THEN

```

```

        DBMS_OUTPUT.PUT_LINE('No employee found with the given ID.');
```

WHEN TOO_MANY_ROWS THEN

```

        DBMS_OUTPUT.PUT_LINE('More than one employee found with the given ID.');
```

WHEN OTHERS THEN

```

        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
```

END;

/

- Explicit Cursors

Explicit cursors in PL/SQL give you more control over the context area created for processing SQL statements. Unlike implicit cursors, which are automatically managed by Oracle, explicit cursors must be declared, opened, fetched, and closed explicitly by the programmer. This allows for more precise handling of query results, especially when dealing with complex queries or when you need to process each row individually. The explicit cursor lifecycle involves four main steps: declaration, opening, fetching, and closing. By managing these steps manually, you can handle multiple rows returned by a query more effectively and perform operations on each row as needed.

Declaring an Explicit Cursor

To declare an explicit cursor, you define it in the declaration section of a PL/SQL block, subprogram, or package. The cursor is associated with a specific SQL query. For example:

```

DECLARE

    CURSOR emp_cursor IS

        SELECT employee_id, employee_name, salary

        FROM employees

        WHERE department_id = 10;
```

In this example, emp_cursor is declared to fetch the employee_id, employee_name, and salary of employees in department 10.

Opening and Fetching from an Explicit Cursor

Once declared, you open the cursor to allocate memory and execute the associated query. You then fetch rows from the cursor into PL/SQL variables. Here's how you can do it:

```

DECLARE

    CURSOR emp_cursor IS

        SELECT employee_id, employee_name, salary

        FROM employees

        WHERE department_id = 10;
```

```

v_employee_id employees.employee_id%TYPE;

v_employee_name employees.employee_name%TYPE;

v_salary employees.salary%TYPE;

BEGIN

OPEN emp_cursor;

LOOP

    FETCH emp_cursor INTO v_employee_id, v_employee_name, v_salary;

    EXIT WHEN emp_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('ID: ' || v_employee_id || ', Name: ' || v_employee_name || ', Salary: ' ||
v_salary);

END LOOP;

CLOSE emp_cursor;

END;

/

```

In this example:

The cursor emp_cursor is opened with the OPEN statement.

The FETCH statement retrieves each row from the cursor into the variables v_employee_id, v_employee_name, and v_salary.

The EXIT WHEN emp_cursor%NOTFOUND statement exits the loop when there are no more rows to fetch.

The cursor is closed with the CLOSE statement after all rows have been processed.

Using Explicit Cursors with Parameters

Explicit cursors can also accept parameters, making them more flexible. Here's an example of a parameterized cursor:

```

DECLARE

CURSOR emp_cursor (p_dept_id NUMBER) IS

    SELECT employee_id, employee_name, salary

    FROM employees

    WHERE department_id = p_dept_id;

v_employee_id employees.employee_id%TYPE;

```

```

v_employee_name employees.employee_name%TYPE;

v_salary employees.salary%TYPE;

BEGIN

OPEN emp_cursor(20); -- Pass department ID as a parameter

LOOP

    FETCH emp_cursor INTO v_employee_id, v_employee_name, v_salary;

    EXIT WHEN emp_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('ID: ' || v_employee_id || ', Name: ' || v_employee_name || ', Salary: ' ||
v_salary);

END LOOP;

CLOSE emp_cursor;

END;

/

```

8. Procedures and Functions

- **Creating Functions**

Oracle PL/SQL functions are powerful subprograms that return a single value and can be used in SQL statements, PL/SQL blocks, or other functions and procedures. Functions are designed to perform specific tasks and return a result, making them highly reusable and efficient for encapsulating business logic. A function in PL/SQL consists of a header and a body. The header includes the function name, parameter list, and return type, while the body contains the declarative section, executable section, and optional exception-handling section. Functions can accept parameters in three modes: `IN` (input), `OUT` (output), and `IN OUT` (both input and output). This flexibility allows functions to process input values and return computed results, making them suitable for a wide range of applications, from simple calculations to complex data transformations.

To create a function, you use the `CREATE OR REPLACE FUNCTION` statement, followed by the function's name, parameters, and return type. The function body is enclosed between the `IS` and `END` keywords. Within the body, you can declare variables, constants, and other PL/SQL constructs, and write the executable code that performs the desired operations. The function must include at least one `RETURN` statement to specify the value to be returned. For example, consider a function that calculates the total sales for a given year:

```

CREATE OR REPLACE FUNCTION get_total_sales(in_year PLS_INTEGER) RETURN NUMBER IS

    l_total_sales NUMBER := 0;

BEGIN

    SELECT SUM(unit_price * quantity)

    INTO l_total_sales

    FROM order_items

    INNER JOIN orders USING (order_id)

```

```

WHERE status = 'Shipped'

AND EXTRACT(YEAR FROM order_date) = in_year;

RETURN l_total_sales;

EXCEPTION

WHEN NO_DATA_FOUND THEN

    RETURN 0;

WHEN OTHERS THEN

    RETURN -1;

END;

/

```

In this example, the function `get_total_sales` takes a year as input and returns the total sales for that year. The `SELECT INTO` statement calculates the total sales by summing the product of `unit_price` and `quantity` for shipped orders in the specified year. The function handles exceptions by returning 0 if no data is found and -1 for any other errors.

Functions can be called in various contexts, such as assignment statements, Boolean expressions, and SQL queries. For instance, you can call the `get_total_sales` function in an assignment statement to store the result in a variable:

```

DECLARE

    l_sales_2017 NUMBER := 0;

BEGIN

    l_sales_2017 := get_total_sales(2017);

    DBMS_OUTPUT.PUT_LINE('Sales 2017: ' || l_sales_2017);

END;

/

```

You can also use functions in SQL queries to filter or compute values dynamically. For example:

```

SELECT employee_id, get_total_sales(2022) AS total_sales

FROM employees

WHERE department_id = 10;

```

- **Creating Procedures**

Oracle PL/SQL procedures are subprograms designed to perform specific actions and can be invoked to execute a series of statements. Unlike functions, which return a single value, procedures do not return values directly but can return multiple values through output parameters. Procedures are useful for encapsulating business logic, performing repetitive tasks, and managing complex operations within the database. They can accept

parameters in three modes: `IN` (input), `OUT` (output), and `IN OUT` (both input and output). This flexibility allows procedures to interact with the calling environment, making them highly versatile for a wide range of applications, from data manipulation to transaction processing.

To create a procedure, you use the `CREATE OR REPLACE PROCEDURE` statement, followed by the procedure's name and parameter list. The procedure body is enclosed between the `IS` (or `AS`) and `END` keywords. Within the body, you can declare variables, constants, and other PL/SQL constructs, and write the executable code that performs the desired operations. Procedures can also include exception-handling sections to manage runtime errors effectively. For example, consider a procedure that updates an employee's salary based on their performance rating:

```
CREATE OR REPLACE PROCEDURE update_salary (  
    p_emp_id IN employees.employee_id%TYPE,  
    p_performance_rating IN NUMBER  
) IS  
BEGIN  
    IF p_performance_rating = 1 THEN  
        UPDATE employees  
        SET salary = salary * 1.10  
        WHERE employee_id = p_emp_id;  
    ELSIF p_performance_rating = 2 THEN  
        UPDATE employees  
        SET salary = salary * 1.05  
        WHERE employee_id = p_emp_id;  
    ELSE  
        DBMS_OUTPUT.PUT_LINE('No salary increase for performance rating: ' ||  
p_performance_rating);  
    END IF;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        DBMS_OUTPUT.PUT_LINE('Employee not found.');
```

```
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
```

```
END;  
  
/
```

In this example, the procedure `update_salary` takes an employee ID and a performance rating as input parameters. Based on the performance rating, it updates the employee's salary by a certain percentage. The procedure includes exception handling to manage cases where the employee is not found or other unexpected errors occur.

Procedures can be called from PL/SQL blocks, other procedures, functions, or even from *SQLPlus*. To call a procedure, you use the *EXECUTE* statement in *SQLPlus* or simply invoke it within a PL/SQL block. For example:

```
BEGIN
    update_salary(100, 1);
END;
/
```

In this example, the `update_salary` procedure is called with an employee ID of 100 and a performance rating of 1, which triggers a 10% salary increase for the specified employee. Procedures can also return multiple values through output parameters, making them suitable for complex operations that require interaction with the calling environment.

Procedures can be grouped into packages for better modularity and reusability. A package is a collection of related procedures, functions, variables, and other PL/SQL constructs. By grouping related subprograms together, packages help organize code logically and make it easier to manage and maintain. For example, you can create a package that includes the `update_salary` procedure along with other related procedures and functions:

```
CREATE OR REPLACE PACKAGE employee_management IS
    PROCEDURE update_salary (
        p_emp_id IN employees.employee_id%TYPE,
        p_performance_rating IN NUMBER
    );
    -- Other procedures and functions can be declared here
END employee_management;
/
```

```
CREATE OR REPLACE PACKAGE BODY employee_management IS
    PROCEDURE update_salary (
        p_emp_id IN employees.employee_id%TYPE,
        p_performance_rating IN NUMBER
    ) IS
    BEGIN
        IF p_performance_rating = 1 THEN
```

```

UPDATE employees

SET salary = salary * 1.10

WHERE employee_id = p_emp_id;

ELSIF p_performance_rating = 2 THEN

UPDATE employees

SET salary = salary * 1.05

WHERE employee_id = p_emp_id;

ELSE

    DBMS_OUTPUT.PUT_LINE('No salary increase for performance rating: ' ||
p_performance_rating);

END IF;

EXCEPTION

WHEN NO_DATA_FOUND THEN

    DBMS_OUTPUT.PUT_LINE('Employee not found.');
```

```

WHEN OTHERS THEN

    DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
```

```

END update_salary;

-- Other procedure and function implementations can be added here

END employee_management;

/
```

In this example, the `employee_management` package includes the `update_salary` procedure, providing a structured way to manage related subprograms. Packages enhance code organization, promote reusability, and improve performance by loading all related subprograms into memory at once.

9. Packages

Oracle PL/SQL packages are powerful constructs that group related procedures, functions, variables, and other PL/SQL elements into a single unit. This modular approach enhances code organization, reusability, and maintainability. A package consists of two parts: the **package specification** and the **package body**. The package specification declares the public elements, such as procedures, functions, and variables, which can be accessed by other PL/SQL code. The package body contains the implementation of these elements, including the actual code for procedures and functions. By separating the interface (specification) from the implementation (body), packages provide a clear structure and allow for easier maintenance and updates. Additionally, packages improve performance by loading all related subprograms into memory at once, reducing the overhead of repeated parsing and execution.

To create a package, you first define the package specification using the `CREATE OR REPLACE PACKAGE` statement. This specification includes declarations of public procedures, functions, and variables. For example, consider a package that manages employee operations:

```

CREATE OR REPLACE PACKAGE employee_management IS

    PROCEDURE update_salary (

        p_emp_id IN employees.employee_id%TYPE,

        p_performance_rating IN NUMBER

    );

    FUNCTION get_total_salary(p_dept_id IN NUMBER) RETURN NUMBER;

    -- Other procedures and functions can be declared here

END employee_management;

/

```

In this example, the employee_management package specification declares a procedure update_salary and a function get_total_salary. These declarations make the subprograms available for use by other PL/SQL code.

Next, you define the package body using the CREATE OR REPLACE PACKAGE BODY statement. The package body contains the implementation of the procedures and functions declared in the specification. For example:

```

CREATE OR REPLACE PACKAGE BODY employee_management IS

    PROCEDURE update_salary (

        p_emp_id IN employees.employee_id%TYPE,

        p_performance_rating IN NUMBER

    ) IS

    BEGIN

        IF p_performance_rating = 1 THEN

            UPDATE employees

            SET salary = salary * 1.10

            WHERE employee_id = p_emp_id;

        ELSIF p_performance_rating = 2 THEN

            UPDATE employees

            SET salary = salary * 1.05

            WHERE employee_id = p_emp_id;

        ELSE

            DBMS_OUTPUT.PUT_LINE('No salary increase for performance rating: ' ||

            p_performance_rating);

        END IF;

    END update_salary;

END employee_management;

```

```

        END IF;

    EXCEPTION

        WHEN NO_DATA_FOUND THEN

            DBMS_OUTPUT.PUT_LINE('Employee not found.');
```

```

        WHEN OTHERS THEN

            DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
```

```

    END update_salary;

FUNCTION get_total_salary(p_dept_id IN NUMBER) RETURN NUMBER IS

    l_total_salary NUMBER := 0;

BEGIN

    SELECT SUM(salary)

    INTO l_total_salary

    FROM employees

    WHERE department_id = p_dept_id;

    RETURN l_total_salary;

EXCEPTION

    WHEN NO_DATA_FOUND THEN

        RETURN 0;

    WHEN OTHERS THEN

        RETURN -1;

END get_total_salary;

-- Other procedure and function implementations can be added here

END employee_management;

/
```

In this example, the package body provides the implementation for the `update_salary` procedure and the `get_total_salary` function. The body includes the executable code that performs the desired operations and handles exceptions.

Packages offer several advantages, including encapsulation, modularity, and improved performance. By grouping related subprograms together, packages help organize code logically and make it easier to manage and maintain. They also allow for better security by controlling access to the package elements. For instance, you can expose

only the necessary procedures and functions in the package specification while keeping the implementation details hidden in the package body. Additionally, packages improve performance by loading all related subprograms into memory at once, reducing the overhead of repeated parsing and execution.

10. Advanced Topics (Optional)

- **Triggers**

Oracle PL/SQL triggers are special types of stored procedures that automatically execute, or “fire,” in response to specific events on a particular table or view. These events can be data manipulation operations such as `INSERT`, `UPDATE`, or `DELETE`. Triggers are used to enforce business rules, maintain audit trails, replicate data, and perform other automated tasks within the database. A trigger consists of a triggering event, a trigger timing (`BEFORE` or `AFTER` the event), and the trigger body, which contains the PL/SQL code to be executed. Triggers can be defined at the row level, where they execute once for each row affected by the triggering event, or at the statement level, where they execute once for the entire SQL statement.

To create a trigger, you use the `CREATE OR REPLACE TRIGGER` statement, specifying the triggering event, timing, and the table or view on which the trigger acts. For example, consider a trigger that automatically updates a `last_updated` column whenever a row in the `employees` table is modified:

```
CREATE OR REPLACE TRIGGER update_last_modified
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    :NEW.last_updated := SYSDATE;
END;
/
```

In this example, the trigger `update_last_modified` fires before any `UPDATE` operation on the `employees` table. The `FOR EACH ROW` clause indicates that the trigger is a row-level trigger, and the `:NEW` keyword refers to the new values of the row being updated. The trigger body sets the `last_updated` column to the current date and time (`SYSDATE`).

Triggers can also be used to enforce complex business rules. For instance, you might want to ensure that an employee’s salary cannot be decreased. Here’s an example of a trigger that enforces this rule:

```
CREATE OR REPLACE TRIGGER prevent_salary_decrease
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    IF :NEW.salary < :OLD.salary THEN
        RAISE_APPLICATION_ERROR(-20001, 'Salary cannot be decreased.');
```

```
    END IF;
END;
```

/

In this example, the trigger `prevent_salary_decrease` fires before any `UPDATE` operation on the `salary` column of the `employees` table. The trigger compares the new salary (`:NEW.salary`) with the old salary (`:OLD.salary`). If the new salary is less than the old salary, the trigger raises an application error, preventing the update.

Triggers can also be used for auditing purposes, such as tracking changes to sensitive data. For example, you can create a trigger that logs changes to the `employees` table into an audit table:

```
CREATE OR REPLACE TRIGGER audit_employee_changes
AFTER INSERT OR UPDATE OR DELETE ON employees
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        INSERT INTO employee_audit (employee_id, action, action_date)
        VALUES (:NEW.employee_id, 'INSERT', SYSDATE);
    ELSIF UPDATING THEN
        INSERT INTO employee_audit (employee_id, action, action_date)
        VALUES (:NEW.employee_id, 'UPDATE', SYSDATE);
    ELSIF DELETING THEN
        INSERT INTO employee_audit (employee_id, action, action_date)
        VALUES (:OLD.employee_id, 'DELETE', SYSDATE);
    END IF;
END;
/
```

In this example, the trigger `audit_employee_changes` fires after any `INSERT`, `UPDATE`, or `DELETE` operation on the `employees` table. The trigger inserts a record into the `employee_audit` table, logging the employee ID, the type of action performed, and the date of the action.

II. Best Practices

- **Code Formatting**

Writing efficient and maintainable PL/SQL code is crucial for ensuring the performance and reliability of your Oracle database applications. Here are some best practices to follow:

I. Use Proper Naming Conventions

Adopting consistent and meaningful naming conventions for variables, procedures, functions, and packages makes your code more readable and maintainable. Use prefixes to indicate the type of variable (e.g., `v_` for variables, `p_` for parameters) and choose descriptive names that convey the purpose of the variable or subprogram.

2. Optimize SQL Statements

Efficient SQL statements are key to good PL/SQL performance. Use indexes appropriately, avoid unnecessary columns in `SELECT` statements, and prefer set-based operations over row-by-row processing. Use the `EXPLAIN PLAN` and `AUTOTRACE` tools to analyze and optimize your SQL queries.

3. Use Bulk Operations

When dealing with large datasets, use bulk operations like `BULK COLLECT` and `FORALL` to minimize context switches between the PL/SQL engine and the SQL engine. This can significantly improve performance by reducing the overhead of multiple SQL statements.

4. Handle Exceptions Properly

Implement robust error handling using exceptions. Use specific exceptions for known error conditions and a generic `WHEN OTHERS` exception handler for unexpected errors. Log errors using a logging mechanism to help with debugging and monitoring.

5. Modularize Code with Procedures and Functions

Break down complex logic into smaller, reusable procedures and functions. This not only makes your code more modular and easier to maintain but also promotes code reuse. Group related procedures and functions into packages for better organization.

6. Use Cursors Efficiently

Use explicit cursors for complex queries that require row-by-row processing. However, avoid using cursors when set-based operations can achieve the same result more efficiently. Close cursors explicitly to free up resources.

7. Document Your Code

Include comments and documentation within your code to explain the purpose and logic of your PL/SQL blocks. This is especially important for complex logic and business rules. Well-documented code is easier to understand and maintain.

8. Test and Debug Thoroughly

Thoroughly test your PL/SQL code to ensure it behaves as expected. Use unit testing frameworks like `utPLSQL` to automate testing. Debug your code using tools like Oracle SQL Developer's debugger to identify and fix issues.

9. Use Bind Variables

Use bind variables in dynamic SQL to improve performance and security. Bind variables help reduce parsing overhead and prevent SQL injection attacks by separating SQL code from data.

10. Follow Coding Standards

Adhere to coding standards and guidelines to ensure consistency across your PL/SQL codebase. This includes indentation, spacing, and naming conventions. Consistent code is easier to read, understand, and maintain.

I2. Resources for Further Learning

- **Books**

- Recommended reading:
 - “Oracle PL/SQL Programming” by Steven Feuerstein
 - “PL/SQL for Dummies” by Michael Rosenblum and Paul Dorsey

- **Online Courses**

- Suggested platforms:
 - Udemy
 - Coursera
 - Oracle Learning Library

- **Community and Forums**

- Where to ask questions and get help:
 - Stack Overflow
 - Oracle Community Forums
 - Reddit’s r/oracle