Chapter 02

PL/SQL Language Fundamentals

Mayko Freitas da Silva

Introduction

Welcome to the world of PL/SQL. In this chapter, you'll dive deep into the essential building blocks that form the foundation of PL/SQL programming. You'll explore the language components in detail, learn about a powerful feature called anchored data types, and master the concepts of scope and visibility in PL/SQL blocks. This knowledge will empower you to write more efficient, maintainable, and robust PL/SQL code within the HR schema and beyond.

In this chapter, you will learn about:

- PL/SQL Language Components
- Anchored Data Types
- Scope of Variables, Blocks, Nested Blocks, and Labels

Lab 2.1: Understanding PL/SQL Building Blocks

After this lab, you will be able to:

- Identify and use PL/SQL language components
- Create valid PL/SQL identifiers
- Understand the role of reserved words and delimiters
- Work with literals and comments in PL/SQL
- Apply best practices in PL/SQL coding

Imagine you're building a house. Just as you need various materials like bricks, wood, and nails, PL/SQL programs are constructed using different components called lexical units. Let's explore these building blocks of PL/SQL in detail.

Character Sets and Lexical Units

PL/SQL supports two character sets: the database character set and the national character set. These character sets include:

1. Letters: A-Z, a-z
2. Digits: 0-9
3. Whitespace characters: space, tab, new line, carriage return
4. Punctuation characters: *, +, -, =, and so on

When you combine these characters, you create lexical units, which are like the words in the English language. The main types of lexical units in PL/SQL are:

1. Identifiers
2. Reserved words
3. Delimiters
4. Literals
5. Comments

Let's examine each of these in detail.

Variables and Identifiers

In PL/SQL, variables are like containers that hold values of a particular data type. Before you can use a variable, you need to declare it. Here's the basic syntax:

```
<variable_name> <data_type> [optional default assignment];
```

When naming your variables (creating identifiers), follow these rules:

- Start with a letter
- Can contain letters, digits, dollar sign ($), number sign (#), and underscore (_)
- Cannot contain spaces or other special characters
- Are not case-sensitive
- Maximum length of 30 characters

Let's look at some examples using the HR schema:

```
DECLARE
  v_employee_id    NUMBER;
  v_first_name     VARCHAR2(20);
  v_salary         employees.salary%TYPE;
  v_hire_date      DATE := SYSDATE;
  v_department_name departments.department_name%TYPE;
BEGIN
  -- Your code here
  NULL;
END;
```

In this example, v_employee_id, v_first_name, v_salary, v_hire_date, and v_department_name are all valid identifiers.

Now, let's look at some invalid identifiers:

```
DECLARE
  2nd_employee     NUMBER;       -- Invalid: Starts with a number
  employee name    VARCHAR2(50);  -- Invalid: Contains a space
  salary$amount    NUMBER;       -- Valid, but not recommended
  CURRENT-DATE     DATE;         -- Invalid: Contains a hyphen
BEGIN
  NULL;
END;
```

Best practice tip: Use meaningful names for your variables and follow a consistent naming convention. For example, you might prefix all variables with "v_" to distinguish them from table columns.

Reserved Words and Delimiters

Reserved words are special words that have a specific meaning in PL/SQL. You can't use them as identifiers. Some examples include:

- BEGIN
- END
- DECLARE
- IF
- THEN
- ELSE
- SELECT
- INSERT
- UPDATE
- DELETE

Delimiters are characters with special meanings, such as:

- Arithmetic operators: +, -, *, /
- Comparison operators: =, <, >, <=, >=, <>
- Assignment operator: :=
- String concatenation operator: ||

Here's an example using both reserved words and delimiters:

```
DECLARE
  v_salary        employees.salary%TYPE;
  v_increased_salary employees.salary%TYPE;
  v_employee_count  NUMBER;
BEGIN
  SELECT salary INTO v_salary
  FROM employees
  WHERE employee_id = 100;

  IF v_salary < 5000 THEN
    v_increased_salary := v_salary * 1.1;
  ELSE
    v_increased_salary := v_salary * 1.05;
  END IF;

  SELECT COUNT(*) INTO v_employee_count
  FROM employees
  WHERE department_id = 60;

  DBMS_OUTPUT.PUT_LINE('New salary: ' || v_increased_salary);
  DBMS_OUTPUT.PUT_LINE('Number of employees in department 60: ' || v_employee_count);
END;
```

In this example, we use reserved words like DECLARE, BEGIN, END, IF, THEN, ELSE, and SELECT. We also use delimiters like <, >, :=, and ||.

Literals and Comments

Literals are fixed values in your code. They can be:

1. Numeric literals: 42, 3.14159
2. Character literals: 'Hello, World!'
3. Boolean literals: TRUE, FALSE
4. Date literals: DATE '2023-05-15'

Comments are notes in your code that the PL/SQL compiler ignores. There are two types:

1. Single-line comments: Start with --
2. Multi-line comments: Enclosed between /* and */

Let's see an example:

```
DECLARE
  v_greeting VARCHAR2(50) := 'Hello, HR department'; -- This is a string literal
  v_pi NUMBER := 3.14159; -- This is a numeric literal
  v_is_active BOOLEAN := TRUE; -- This is a Boolean literal
  v_hire_date DATE := DATE '2023-01-01'; -- This is a date literal
BEGIN
  /* This is a multi-line comment
     It can span several lines */
  DBMS_OUTPUT.PUT_LINE(v_greeting);
  DBMS_OUTPUT.PUT_LINE('Pi value: ' || v_pi);
  DBMS_OUTPUT.PUT_LINE('Hire date: ' || v_hire_date);
END;
```

Best practice tip: Use comments to explain complex logic or to provide context for your code. However, avoid over-commenting obvious operations.

Lab 2.2: Exploring Anchored Data Types

After this lab, you will be able to:

- Understand and use anchored data types
- Create variables based on table column definitions
- Appreciate the benefits of using anchored data types

Anchored data types are like genetic inheritance in programming. Just as children inherit traits from their parents, variables can inherit characteristics from other variables or table columns. This feature is particularly useful when working with database tables, as it ensures that your PL/SQL variables always match the data types of the corresponding table columns.

The syntax for an anchored declaration is:

```
<variable_name> <type_attribute>%TYPE;
```

Let's see some examples using the HR schema:

```
DECLARE
  v_last_name employees.last_name%TYPE;
  v_hire_date employees.hire_date%TYPE;
  v_salary    employees.salary%TYPE;
  v_job_id    employees.job_id%TYPE;
  v_dept_name departments.department_name%TYPE;
BEGIN
  SELECT last_name, hire_date, salary, job_id
  INTO v_last_name, v_hire_date, v_salary, v_job_id
  FROM employees
  WHERE employee_id = 100;

  SELECT department_name
  INTO v_dept_name
  FROM departments
  WHERE department_id = 60;

  DBMS_OUTPUT.PUT_LINE('Employee: ' || v_last_name);
  DBMS_OUTPUT.PUT_LINE('Hired: ' || v_hire_date);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
  DBMS_OUTPUT.PUT_LINE('Job ID: ' || v_job_id);
  DBMS_OUTPUT.PUT_LINE('Department: ' || v_dept_name);
END;
```

In this example, all variables inherit their data types from the corresponding table columns. This approach offers several benefits:

1. Automatic data type matching: If the column's data type changes in the database, your PL/SQL code doesn't need to be updated.
2. Reduced maintenance: You don't need to manually specify and update data types.
3. Improved code reliability: It eliminates errors that could occur from mismatched data types.

You can also use the %TYPE attribute to base a variable's data type on another variable:

```
DECLARE
  v_salary    employees.salary%TYPE;
  v_bonus     v_salary%TYPE;
BEGIN
  SELECT salary INTO v_salary
  FROM employees
  WHERE employee_id = 100;

  v_bonus := v_salary * 0.1;

  DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
  DBMS_OUTPUT.PUT_LINE('Bonus: ' || v_bonus);
END;
```

In this case, v_bonus inherits its data type from v_salary, which in turn inherits from the employees.salary column.

Best practice tip: Use anchored data types whenever possible, especially when working with database tables. This practice will make your code more maintainable and less prone to errors.

Lab 2.3: Mastering Scope and Visibility

After this lab, you will be able to:

- Understand variable scope and visibility
- Work with nested blocks
- Use labels to improve code readability and functionality
- Apply scope and visibility concepts to write more organized code

Think of scope and visibility like rooms in a house. Some items (variables) are visible from anywhere in the house (global scope), while others are only visible in specific rooms (local scope).

Variable Scope

The scope of a variable is the part of the program where the variable can be accessed. It typically extends from the point of declaration to the end of the block where it was declared.

Let's look at an example:

```
DECLARE
  v_global_var NUMBER := 100;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Global variable: ' || v_global_var);

  DECLARE
    v_local_var NUMBER := 200;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Local variable: ' || v_local_var);
    DBMS_OUTPUT.PUT_LINE('Global variable inside inner block: ' || v_global_var);
  END;

  DBMS_OUTPUT.PUT_LINE('Global variable after inner block: ' || v_global_var);
  -- This would cause an error because v_local_var is not visible here
  -- DBMS_OUTPUT.PUT_LINE('Trying to access local variable: ' || v_local_var);
END;
```

In this example, v_global_var is visible throughout the entire block, while v_local_var is only visible within the inner block where it's declared.

Nested Blocks

PL/SQL allows you to nest blocks within each other. This is useful for organizing your code and controlling variable scope. Let's see a more complex example using the HR schema:

```
DECLARE
 v_dept_id NUMBER := 60;
 v_dept_name departments.department_name%TYPE;
BEGIN
 -- Outer block
 SELECT department_name INTO v_dept_name
 FROM departments
 WHERE department_id = v_dept_id;

 DBMS_OUTPUT.PUT_LINE('Department: ' || v_dept_name);

 DECLARE
  v_emp_count NUMBER;
  v_avg_salary employees.salary%TYPE;
 BEGIN
  -- Inner block
  SELECT COUNT(*), AVG(salary)
  INTO v_emp_count, v_avg_salary
  FROM employees
  WHERE department_id = v_dept_id;

  DBMS_OUTPUT.PUT_LINE('Number of employees: ' || v_emp_count);
  DBMS_OUTPUT.PUT_LINE('Average salary: ' || v_avg_salary);

  DECLARE
   v_max_salary employees.salary%TYPE;
  BEGIN
   -- Innermost block
   SELECT MAX(salary)
   INTO v_max_salary
   FROM employees
   WHERE department_id = v_dept_id;

   DBMS_OUTPUT.PUT_LINE('Maximum salary: ' || v_max_salary);
  END;
 END;
END;
```

In this example, we have three levels of nested blocks. Each inner block can access variables from outer blocks, but outer blocks cannot access variables declared in inner blocks.

Using Labels

Labels can improve code readability and allow you to reference variables from outer blocks. Here's an example:

```
<<outer_block>>
DECLARE
  v_employee_id NUMBER := 100;
  v_last_name employees.last_name%TYPE;
BEGIN
  SELECT last_name INTO v_last_name
  FROM employees
  WHERE employee_id = v_employee_id;

  DBMS_OUTPUT.PUT_LINE('Outer employee ID: ' || v_employee_id);
  DBMS_OUTPUT.PUT_LINE('Employee name: ' || v_last_name);

  <<inner_block>>
  DECLARE
    v_employee_id NUMBER := 101;
    v_first_name employees.first_name%TYPE;
  BEGIN
    SELECT first_name INTO v_first_name
    FROM employees
    WHERE employee_id = v_employee_id;

    DBMS_OUTPUT.PUT_LINE('Inner employee ID: ' || v_employee_id);
    DBMS_OUTPUT.PUT_LINE('Inner employee first name: ' || v_first_name);
    DBMS_OUTPUT.PUT_LINE('Outer employee ID: ' || outer_block.v_employee_id);
    DBMS_OUTPUT.PUT_LINE('Outer employee last name: ' || outer_block.v_last_name);
  END inner_block;
END outer_block;
```

In this example, we use labels to distinguish between the v_employee_id variables in the outer and inner blocks. The label allows us to access the outer block's variable even when it's shadowed by a variable with the same name in the inner block.

Best practice tip: Use labels to improve code readability, especially in complex nested blocks. However, be cautious not to overuse them, as it can make the code harder to follow.

Summary

In this chapter, you've gained a comprehensive understanding of PL/SQL language fundamentals. You've learned about the various components that make up PL/SQL code, including character sets, lexical units, variables, identifiers, reserved words, delimiters, literals, and comments. You've explored the power of anchored data types, which allow variables to inherit characteristics from other variables or table columns, improving code maintainability and reducing errors.

You've also mastered the concepts of scope and visibility, learning how to use nested blocks and labels to organize your code effectively. These skills will allow you to write more structured, efficient, and maintainable PL/SQL code in the HR schema and beyond.

Remember to apply best practices such as using meaningful variable names, leveraging anchored data types, and using comments and labels judiciously to improve code readability and maintainability.

As you continue your journey with PL/SQL, these foundational concepts will serve as the building blocks for more advanced techniques and complex programming tasks.

PL/SQL Language Fundamentals