**Introduction:**

PL/SQL is Oracle's procedural extension to SQL, combining the power of SQL with the programming features of a procedural language. It was developed to enhance SQL's capabilities, allowing for more complete and complex programming solutions within the Oracle Database.

**Purpose and Features 🎯**

The core purpose of PL/SQL is to enable developers to merge database commands with procedural programming logic. This allows for the creation of robust applications that can handle complex business logic directly within the database.

Key features of PL/SQL include:

* **Tight integration with SQL**: It allows you to embed SQL statements directly within your PL/SQL code.
* **Error handling**: It provides extensive error-checking mechanisms to ensure program stability.
* **Rich data types and structures**: It supports a wide range of data types and programming structures like loops and conditionals.
* **Support for modular and object-oriented programming**: It enables the use of procedures, functions, and object types to create structured and reusable code.

**PL/SQL Block Structure 🧱**

PL/SQL programs are built using a block-structured format. Every block consists of three main sections, though only one is mandatory.

1. **DECLARE (Optional)**: This section is for declaring variables, cursors, and other elements needed within the block.
2. **BEGIN...END (Mandatory)**: This is the executable part of the block, containing all the procedural and SQL statements. It must have at least one executable line.
3. **EXCEPTION (Optional)**: This section is used for handling errors and exceptions that may occur during the execution of the program.

All PL/SQL statements are terminated with a semicolon. Blocks can also be nested within each other to create more complex program flows.

**Applications and Career Opportunities 💼**

PL/SQL is widely used in various applications, including database security, XML management, web development, and database automation. Its proficiency is a valuable skill for database professionals.

Learning PL/SQL can lead to several career paths, such as:

* Oracle PL/SQL Programmer
* Database Developer
* Data Analyst
* ETL Developer
* Database Migration Expert

Its ease of use, portability across Oracle databases, high performance, and security features make it an essential skill for anyone involved in database management and development.

**Overview:**

PL/SQL, which stands for **Procedural Language/Structured Query Language**, is Oracle's procedural extension to SQL. Developed in the late 1980s, it combines the data manipulation power of SQL with the procedural capabilities of a programming language, allowing for more complex business logic to be executed directly within the database. It is **portable**, **high-performance**, and has a syntax based on Ada and Pascal.

**Features and Advantages**

PL/SQL offers a rich set of features that provide significant advantages for database development:

* **Tight SQL Integration**: PL/SQL is strongly integrated with SQL, supporting both **static** (DML and transaction control) and **dynamic** (DDL) SQL within its blocks. This allows for efficient data manipulation and transformation.
* **High Performance**: By sending an entire block of statements to the database at once, PL/SQL minimizes network traffic and improves application performance.
* **Enhanced Productivity**: It allows developers to quickly create robust applications by providing features like EXCEPTION handling, **encapsulation**, and **data hiding**, which simplifies debugging and design.
* **Security and Portability**: PL/SQL provides a high level of security to protect database integrity. Programs written in PL/SQL are also highly portable across different Oracle databases.
* **Object-Oriented Support**: It supports **object-oriented programming** concepts, allowing for more structured and reusable code through the use of functions, procedures, and object types.
* **Web Development**: PL/SQL can be used for developing web applications and server pages, linking databases directly to web frontends.

**Basic Syntax:**

PL/SQL's basic syntax is **block-structured**, meaning code is organized into logical, self-contained units. Each block is composed of three key sections: a declaration section, an executable commands section, and an optional exception-handling section.

**PL/SQL Block Structure 🧱**

The standard structure of a PL/SQL block is as follows:

Code snippet

DECLARE

-- Optional: Declarations of variables, cursors, etc.

BEGIN

-- Mandatory: Executable code (at least one line)

EXCEPTION

-- Optional: Error handling statements

END;

/

* **DECLARE**: This section is where you define any variables, constants, or other elements that will be used in the block. It's optional.
* **BEGIN...END**: This is the **mandatory** section containing the main executable code. It must contain at least one statement, even if it's just NULL.
* **EXCEPTION**: This optional section is where you place code to handle specific errors or exceptions that might occur during execution.

Every PL/SQL statement must end with a semicolon (;). The END; statement concludes the block, and often, a / on a new line is used in SQL\*Plus to execute the block.

**Identifiers and Delimiters ✍️**

**Identifiers** in PL/SQL are names given to variables, constants, procedures, and other program elements. They can be up to 30 characters long and are **case-insensitive** by default. They must start with a letter and can include letters, numbers, dollar signs ($), underscores (\_), and number signs (#). You cannot use a reserved keyword as an identifier.

**Delimiters** are symbols with special meanings that are used to separate elements in PL/SQL code. Examples include:

* +, -, \*, /: Arithmetic operators
* :=: Assignment operator
* --: Single-line comment
* /\* ... \*/: Multi-line comment
* ;: Statement terminator

Comments are explanatory notes ignored by the compiler. Single-line comments start with --, while multi-line comments are enclosed in /\* and \*/.

|  |  |
| --- | --- |
| **Delimiter** | **Description** |
| **+, -, \*, /** | Addition, subtraction/negation, multiplication, division |
| **%** | Attribute indicator |
| **'** | Character string delimiter |
| **.** | Component selector |
| **(,)** | Expression or list delimiter |
| **:** | Host variable indicator |
| **,** | Item separator |
| **"** | Quoted identifier delimiter |
| **=** | Relational operator |
| **@** | Remote access indicator |
| **;** | Statement terminator |
| **:=** | Assignment operator |
| **=>** | Association operator |
| **||** | Concatenation operator |
| **\*\*** | Exponentiation operator |
| **<<, >>** | Label delimiter (begin and end) |
| **/\*, \*/** | Multi-line comment delimiter (begin and end) |
| **--** | Single-line comment indicator |
| **..** | Range operator |
| **<, >, <=, >=** | Relational operators |
| **<>, '=, ~=, ^=** | Different versions of NOT EQUAL |

**PL/SQL Program Units 📦**

A PL/SQL unit refers to any of the fundamental building blocks of a PL/SQL application. These units, which can be stored in the database, include:

* A standalone **PL/SQL block**
* **Functions** and **Procedures**
* **Packages** and **Package bodies**
* **Triggers**
* **Types** and **Type bodies**

**DATA TYPES :**

In PL/SQL, variables, constants, and parameters must be assigned a **data type**, which defines the storage format, constraints, and valid range of values. PL/SQL organizes these data types into four categories: Scalar, Large Object (LOB), Composite, and Reference. This overview focuses on the Scalar and LOB types.

**1. Scalar Data Types 🔢**

Scalar data types hold single, atomic values. PL/SQL categorizes them into four main groups:

* **Numeric**: Used for numbers on which you can perform arithmetic operations. Examples include NUMBER, INTEGER, PLS\_INTEGER, and floating-point types like BINARY\_FLOAT and BINARY\_DOUBLE. NUMBER(prec, scale) provides specific precision and scale.

|  |  |
| --- | --- |
| **S.No** | **Data Type & Description** |
| 1 | **PLS\_INTEGER**  Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| 2 | **BINARY\_INTEGER**  Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| 3 | **BINARY\_FLOAT**  Single-precision IEEE 754-format floating-point number |
| 4 | **BINARY\_DOUBLE**  Double-precision IEEE 754-format floating-point number |
| 5 | **NUMBER(prec, scale)**  Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0 |
| 6 | **DEC(prec, scale)**  ANSI specific fixed-point type with maximum precision of 38 decimal digits |
| 7 | **DECIMAL(prec, scale)**  IBM specific fixed-point type with maximum precision of 38 decimal digits |
| 8 | **NUMERIC(pre, secale)**  Floating type with maximum precision of 38 decimal digits |
| 9 | **DOUBLE PRECISION**  ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits) |
| 10 | **FLOAT**  ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits) |
| 11 | **INT**  ANSI specific integer type with maximum precision of 38 decimal digits |
| 12 | **INTEGER**  ANSI and IBM specific integer type with maximum precision of 38 decimal digits |
| 13 | **SMALLINT**  ANSI and IBM specific integer type with maximum precision of 38 decimal digits |
| 14 | **REAL**  Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits) |

* **Character**: Used for strings of characters. Common types are CHAR (fixed-length), VARCHAR2 (variable-length), and NCHAR (fixed-length national character set).

|  |  |
| --- | --- |
| **S.No** | **Data Type & Description** |
| 1 | **CHAR**  Fixed-length character string with maximum size of 32,767 bytes |
| 2 | **VARCHAR2**  Variable-length character string with maximum size of 32,767 bytes |
| 3 | **RAW**  Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL |
| 4 | **NCHAR**  Fixed-length national character string with maximum size of 32,767 bytes |
| 5 | **NVARCHAR2**  Variable-length national character string with maximum size of 32,767 bytes |
| 6 | **LONG**  Variable-length character string with maximum size of 32,760 bytes |
| 7 | **LONG RAW**  Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL |
| 8 | **ROWID**  Physical row identifier, the address of a row in an ordinary table |
| 9 | **UROWID**  Universal row identifier (physical, logical, or foreign row identifier) |

* **Boolean**: Stores logical values: TRUE, FALSE, or NULL. This type is unique to PL/SQL and has no direct equivalent in SQL, meaning you cannot use Boolean variables in SQL statements.
* **Datetime**: Used for storing dates and times. The primary type is DATE, which includes the century, year, month, day, hour, minute, and second. Other types like TIMESTAMP offer higher precision with fractional seconds.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Valid Datetime Values** | **Valid Interval Values** |
| YEAR | -4712 to 9999 (excluding year 0) | Any nonzero integer |
| MONTH | 01 to 12 | 0 to 11 |
| DAY | 01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale) | Any nonzero integer |
| HOUR | 00 to 23 | 0 to 23 |
| MINUTE | 00 to 59 | 0 to 59 |
| SECOND | 00 to 59.9(n), where 9(n) is the precision of time fractional seconds | 0 to 59.9(n), where 9(n) is the precision of interval fractional seconds |
| TIMEZONE\_HOUR | -12 to 14 (range accommodates daylight savings time changes) | Not applicable |
| TIMEZONE\_MINUTE | 00 to 59 | Not applicable |
| TIMEZONE\_REGION | Found in the dynamic performance view V$TIMEZONE\_NAMES | Not applicable |
| TIMEZONE\_ABBR | Found in the dynamic performance view V$TIMEZONE\_NAMES | Not applicable |

**2. Large Object (LOB) Data Types 📂**

LOB data types are designed to handle very large, unstructured data. Instead of storing the data directly within the table row, LOBs store a pointer to the data, which is held separately.

* **BFILE**: Stores large binary objects in operating system files outside the database. It can hold up to 4 gigabytes (GB).
* **BLOB**: Stores large binary objects directly within the database. The size limit is 8 to 128 terabytes (TB).
* **CLOB**: Stores large blocks of character data within the database. It is ideal for large text documents and has the same size limit as BLOB.
* **NCLOB**: Stores large blocks of national character set data within the database.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Size** |
| BFILE | Used to store large binary objects in operating system files outside the database. | System-dependent. Cannot exceed 4 gigabytes (GB). |
| BLOB | Used to store large binary objects in the database. | 8 to 128 terabytes (TB) |
| CLOB | Used to store large blocks of character data in the database. | 8 to 128 TB |
| NCLOB | Used to store large blocks of NCHAR data in the database. | 8 to 128 TB |

**PL/SQL Subtypes and NULL Values 🤔**

* **Subtypes**: A subtype is a constrained version of another data type (its base type). For example, INTEGER is a subtype of NUMBER. You can also define your own subtypes to improve code readability and compatibility.
* **NULL**: In PL/SQL, NULL represents a missing or unknown value. It is not equivalent to an empty string or zero, and a NULL value can't be equated with anything, including itself.

**VARIABLES :**

In PL/SQL, a **variable** is a named storage area that holds a value of a specific data type. Declaring a variable tells the PL/SQL compiler to reserve memory for that variable. Variable names are **case-insensitive** by default, must begin with a letter, and can include numbers, \_, $, and #, with a maximum length of 30 characters. You can't use a reserved keyword as a variable name.

**Variable Declaration and Initialization 📝**

Variables must be declared in the DECLARE section of a PL/SQL block or as a global variable within a package.

The basic syntax for a variable declaration is:

variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial\_value]

* **variable\_name**: The name of your variable.
* **datatype**: A valid PL/SQL data type (e.g., NUMBER, VARCHAR2, INTEGER).
* **CONSTANT**: An optional keyword to declare a constant, which requires an initial value.
* **NOT NULL**: An optional constraint that prevents the variable from holding a NULL value. If used, you must assign an initial value.

**Initialization**: When a variable is declared, its default value is NULL. You can provide an initial value using either the **assignment operator (:=)** or the **DEFAULT keyword**.

Code snippet

DECLARE

-- Declaring and initializing with assignment operator

counter INTEGER := 0;

-- Declaring and initializing with the DEFAULT keyword

greetings VARCHAR2(20) DEFAULT 'Hello!';

-- Declaring a constant

pi CONSTANT NUMBER := 3.14;

BEGIN

NULL; -- A mandatory executable statement

END;

/

**Constrained vs. Unconstrained Declarations**: Providing a size or precision with a data type (e.g., VARCHAR2(25)) is a **constrained declaration**. This uses less memory than an unconstrained one and is a good practice for memory efficiency.

**Variable Scope 🔭**

PL/SQL supports nested blocks, which affects variable scope.

* **Global Variables**: Variables declared in an outer block are **global** and accessible to all inner, nested blocks.
* **Local Variables**: Variables declared inside an inner block are **local** and are only accessible within that block.

If an inner block declares a variable with the same name as a global variable, the inner block's variable **hides** the global one within its scope.

Code snippet

DECLARE

-- Global variable

num1 NUMBER := 95;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Outer Variable num1: ' || num1);

DECLARE

-- Local variable

num1 NUMBER := 195;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Inner Variable num1: ' || num1);

END;

END;

/

**Output:**

Outer Variable num1: 95

Inner Variable num1: 195

**Assigning Values from SQL Queries 📊**

You can assign the result of an SQL query to PL/SQL variables using the **SELECT INTO** statement. This is a common method for fetching data from a database table into program variables.

The SELECT list must have the same number of columns as the variables in the INTO list, and their data types must be compatible.

Code snippet

DECLARE

-- c\_name is a variable with the same data type as the NAME column in the customers table

c\_name customers.name%TYPE;

BEGIN

-- Fetches the name from the customers table where ID is 1 and assigns it to c\_name

SELECT name INTO c\_name

FROM customers

WHERE id = 1;

DBMS\_OUTPUT.PUT\_LINE('Customer Name: ' || c\_name);

END;

/

**CONSTANTS & LITERALS :**

In PL/SQL, a **constant** is a variable whose value is set at declaration and cannot be changed later in the program. A **literal** is a fixed, explicit value that is not represented by a name.

**Constants 🧊**

Constants are declared using the **CONSTANT** keyword. They must be initialized with a value at the time of declaration, and this value remains fixed for the duration of the program's execution. Using constants improves code readability and maintainability by allowing you to give a meaningful name to a fixed value, like PI.

**Syntax:**

Code snippet

constant\_name CONSTANT datatype := value;

**Example:**

Code snippet

DECLARE

-- Declare a constant named pi

pi CONSTANT NUMBER := 3.14159;

radius NUMBER := 5;

area NUMBER;

BEGIN

area := pi \* radius \* radius;

DBMS\_OUTPUT.PUT\_LINE('Area of circle: ' || area);

END;

/

In this example, the value of pi cannot be modified anywhere else in the block.

**Literals 🔡**

A literal is a raw, explicit value written directly in your code. They are not stored in variables and have a specific data type determined by their format. Unlike most identifiers in PL/SQL, string and character literals are **case-sensitive**.

PL/SQL supports several types of literals:

* **Numeric Literals**: These are numbers. They can be integers (50), fixed-point (6.66), or floating-point with an exponent (6E5).
* **Character Literals**: A single character enclosed in single quotes, e.g., 'A'.
* **String Literals**: A sequence of characters enclosed in single quotes, e.g., 'Hello, World!'. To include a single quote within a string, use two single quotes ('').
* **Boolean Literals**: The logical values TRUE, FALSE, and NULL.
* **Date and Time Literals**: These are used to represent specific moments in time, for example, DATE '1978-12-25' or TIMESTAMP '2012-10-29 12:01:01'.

**OPERATORS :**

An operator in PL/SQL is a symbol that tells the compiler to perform a specific action, such as a mathematical or logical operation. PL/SQL offers a variety of built-in operators that can be grouped into several categories.

**Arithmetic Operators ➕➖**

These operators perform mathematical calculations.

| Operator | Description | Example (A=10, B=5) |
| --- | --- | --- |
| + | Adds two operands. | A + B is 15 |
| - | Subtracts the second operand from the first. | A - B is 5 |
| \* | Multiplies both operands. | A \* B is 50 |
| / | Divides the numerator by the denominator. | A / B is 2 |
| \*\* | Exponentiation. Raises the first operand to the power of the second. | A \*\* B is 100000 |

**Relational Operators ⚖️**

These operators compare two expressions and return a **Boolean** (TRUE, FALSE, or NULL) result.

| Operator | Description | Example (A=10, B=20) |
| --- | --- | --- |
| = | Checks if operands are equal. | (A = B) is FALSE |
| !=, <>, ~= | Checks if operands are not equal. | (A != B) is TRUE |
| > | Checks if the left operand is greater than the right. | (A > B) is FALSE |
| < | Checks if the left operand is less than the right. | (A < B) is TRUE |
| >= | Checks if the left operand is greater than or equal to the right. | (A >= B) is FALSE |
| <= | Checks if the left operand is less than or equal to the right. | (A <= B) is TRUE |

**Comparison Operators 🕵️**

These operators are used for comparing an expression to a value or a range.

* **LIKE**: Compares a string to a pattern. TRUE if the string matches.
  + Example: 'Zara Ali' LIKE 'Z% A\_i' returns TRUE.
* **BETWEEN**: Tests if a value is within a specified range (inclusive).
  + Example: 10 BETWEEN 5 AND 20 returns TRUE.
* **IN**: Tests if a value is a member of a specified set.
  + Example: 'm' IN ('m', 'n', 'o') returns TRUE.
* **IS NULL**: Tests if a value is NULL. Note that comparisons with NULL always return NULL unless using this operator.

**Logical Operators 🧠**

These operators work on Boolean operands to produce a single Boolean result.

| Operator | Description | Example (A=TRUE, B=FALSE) |
| --- | --- | --- |
| AND | Returns TRUE if both operands are TRUE. | (A AND B) is FALSE |
| OR | Returns TRUE if at least one operand is TRUE. | (A OR B) is TRUE |
| NOT | Reverses the logical state of the operand. | NOT (A AND B) is TRUE |

**Operator Precedence ⛓️**

Operator precedence determines the order in which an expression is evaluated. Operators with higher precedence are evaluated before those with lower precedence. For example, multiplication (\*) and division (/) have a higher precedence than addition (+) and subtraction (-).

**Precedence Order (highest to lowest):**

1. \*\* (Exponentiation)
2. +, - (Identity, negation)
3. \*, / (Multiplication, division)
4. +, -, || (Addition, subtraction, concatenation)
5. Comparison operators (=, <, >, <=, >=, <>, !=, etc.)
6. NOT (Logical negation)
7. AND (Conjunction)
8. OR (Inclusion)

**CONDITIONAL STATEMENTS :**

In PL/SQL, **conditional statements** control the flow of a program by executing specific blocks of code based on whether a condition is TRUE or FALSE. These structures are essential for decision-making within your code.

A diagram of a condition

AI-generated content may be incorrect.

**1. IF-THEN Statement**

This is the simplest conditional statement. If a **condition** is TRUE, the statements within the block are executed. If the condition is FALSE or NULL, the block is skipped entirely.

**Syntax:**

Code snippet

IF condition THEN

-- statements to execute if condition is TRUE

END IF;

**2. IF-THEN-ELSE Statement**

This statement provides an alternative path for execution. If the **condition** is TRUE, the code in the THEN block runs. If it's FALSE or NULL, the code in the ELSE block is executed. This ensures that one of the two blocks is always run.

**Syntax:**

Code snippet

IF condition THEN

-- statements to execute if condition is TRUE

ELSE

-- statements to execute if condition is FALSE

END IF;

**3. IF-THEN-ELSIF Statement**

Used for handling multiple, mutually exclusive conditions. This structure checks conditions sequentially. The first ELSIF or IF condition that evaluates to TRUE has its block of statements executed, and then the entire IF statement is terminated. The optional ELSE block at the end acts as a catch-all for any conditions that didn't match.

**Syntax:**

Code snippet

IF condition1 THEN

-- statements for condition1

ELSIF condition2 THEN

-- statements for condition2

ELSIF condition3 THEN

-- statements for condition3

ELSE

-- optional statements if no conditions are met

END IF;

**4. CASE Statement**

The CASE statement selects one of several alternatives to execute based on the value of a single **selector expression**. This is often a cleaner alternative to a long IF-THEN-ELSIF chain when you are checking a single variable against multiple possible values.

**Syntax:**

Code snippet

CASE selector

WHEN value1 THEN

-- statements for value1

WHEN value2 THEN

-- statements for value2

...

ELSE

-- optional default statements

END CASE;

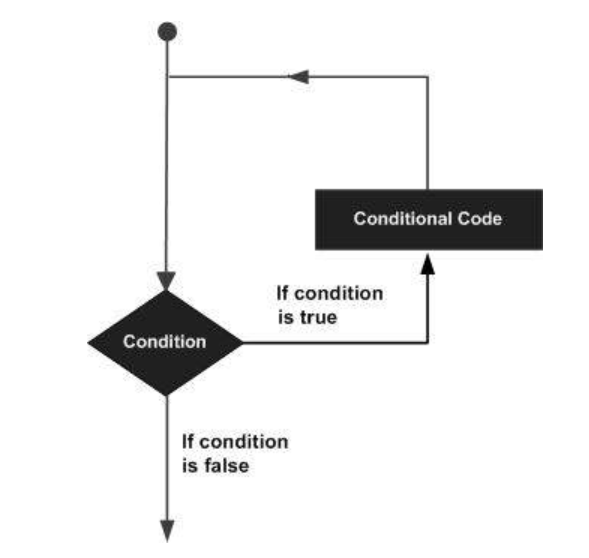
A variation, the **searched CASE statement**, works without a selector and uses boolean expressions in its WHEN clauses, similar to an IF-THEN-ELSIF but often more readable.

**5. Nested IF-THEN-ELSE Statements**

PL/SQL allows you to nest IF statements inside other IF statements. This is useful for handling complex, multi-level conditions. However, excessive nesting can make code difficult to read and debug, so it's often better to use IF-THEN-ELSIF or a CASE statement when possible.

**LOOPS :**

In PL/SQL, **loops** are control structures that allow a block of code to be executed repeatedly. There are three main types of loops: BASIC LOOP, WHILE LOOP, and FOR LOOP. PL/SQL also supports **nested loops** and **loop labels** for more complex control flow.



**Types of Loops 🔁**

1. **BASIC LOOP**: The simplest form of loop. It executes a sequence of statements repeatedly and requires an explicit EXIT or EXIT WHEN statement to terminate. If there is no exit condition, it will run indefinitely.

Code snippet

LOOP

-- statements to execute

EXIT WHEN condition; -- Exit condition

END LOOP;

1. **WHILE LOOP**: An entry-controlled loop that executes a block of code as long as a specified condition is true. The condition is checked at the beginning of each iteration.

Code snippet

WHILE condition LOOP

-- statements to execute

END LOOP;

1. **FOR LOOP**: A compact and powerful loop used for executing a sequence of statements a predetermined number of times. It automatically handles the loop counter and its increment/decrement.

Code snippet

FOR counter IN [REVERSE] lower\_bound .. upper\_bound LOOP

-- statements to execute

END LOOP;

**Loop Control Statements 🚀**

Loop control statements alter the normal flow of a loop.

* **EXIT**: Immediately terminates the current loop and transfers control to the next statement after END LOOP. It can be used with a condition (EXIT WHEN condition;).
* **CONTINUE**: Skips the rest of the statements in the current iteration of the loop and immediately starts the next iteration.
* **GOTO**: Transfers control to a specific labeled statement. While it's a valid feature, its use is generally discouraged as it can lead to complex and hard-to-read code.

**Labeling Loops 🏷️**

PL/SQL allows you to label loops using <<label\_name>>. This is particularly useful for **nested loops**, as it allows you to specify which loop to exit or continue.

Code snippet

<< outer\_loop >>

FOR i IN 1..3 LOOP

<< inner\_loop >>

FOR j IN 1..3 LOOP

EXIT outer\_loop WHEN i = 2 AND j = 2; -- Exits both loops

DBMS\_OUTPUT.PUT\_LINE('i: ' || i || ', j: ' || j);

END LOOP inner\_loop;

END LOOP outer\_loop;

**STRINGS :**

In PL/SQL, a **string** is a sequence of characters that can be a combination of letters, numbers, and special characters. PL/SQL supports three main types of strings: fixed-length, variable-length, and Character Large Objects (CLOBs).

**String Types and Declaration 📝**

1. **Fixed-length strings**: The length is specified during declaration using the CHAR datatype. If the assigned string is shorter, it's automatically padded with spaces to the specified length.

Code snippet

my\_char CHAR(10); -- A fixed-length string of 10 characters

1. **Variable-length strings**: These can have a maximum length up to 32,767 bytes and are declared using the VARCHAR2 datatype. No extra padding is added.

Code snippet

my\_varchar VARCHAR2(50); -- A variable-length string up to 50 chars

1. **Character Large Objects (CLOBs)**: Used for storing very large amounts of character data, up to 128 terabytes.

Code snippet

my\_clob CLOB;

Oracle also provides national character set types like NCHAR, NVARCHAR2, and NCLOB to store Unicode character data.

**String Operators and Functions ⚙️**

PL/SQL provides a rich set of built-in functions for manipulating strings. The primary operator is the **concatenation operator (||)**, which joins two strings together.

**Common String Functions:**

| Function | Purpose | Example |
| --- | --- | --- |
| **UPPER**, **LOWER** | Convert a string to all uppercase or lowercase. | UPPER('hello') returns 'HELLO' |
| **INITCAP** | Capitalizes the first letter of each word in a string. | INITCAP('hello world') returns 'Hello World' |
| **LENGTH** | Returns the number of characters in a string. | LENGTH('apple') returns 5 |
| **SUBSTR** | Extracts a substring from a string. | SUBSTR('example', 3, 4) returns 'ampl' |
| **INSTR** | Returns the starting position of a substring within a string. | INSTR('hello', 'o') returns 5 |
| **TRIM** | Removes leading and/or trailing characters from a string. | TRIM('x' FROM 'xxheyxx') returns 'hey' |
| **REPLACE** | Replaces all occurrences of a substring with another string. | REPLACE('one two', 'two', 'three') returns 'one three' |
| **CONCAT** | Concatenates two strings. | CONCAT('hello', 'world') returns 'helloworld' |

**Example:**

Code snippet

DECLARE

greetings VARCHAR2(20) := 'hello world';

BEGIN

-- Concatenation

DBMS\_OUTPUT.PUT\_LINE('The string is: ' || greetings);

-- Using string functions

DBMS\_OUTPUT.PUT\_LINE(UPPER(greetings)); -- HELLO WORLD

DBMS\_OUTPUT.PUT\_LINE(SUBSTR(greetings, 7, 5)); -- world

END;

/

**ARRAY :**

In PL/SQL, a **VARRAY** (variable-size array) is a data structure used to store a fixed-size, sequential collection of elements of the same data type. It is a one-dimensional array where elements are stored in contiguous memory locations and are accessed by their index, which starts at 1.

**Creating a VARRAY Type ✍️**

To use a VARRAY, you must first define its type. This can be done either at the schema level or within a PL/SQL block.

1. **Schema-Level VARRAY**: Use the CREATE OR REPLACE TYPE statement to define a VARRAY type that can be reused across different programs.

**Syntax:**

Code snippet

CREATE OR REPLACE TYPE varray\_type\_name IS VARRAY(n) OF <element\_type>;

* + n: The maximum number of elements the array can hold.
  + <element\_type>: The data type of the elements (e.g., VARCHAR2, INTEGER).

1. **VARRAY within a PL/SQL Block**: Define the VARRAY type directly within the DECLARE section of a block.

**Syntax:**

Code snippet

TYPE varray\_type\_name IS VARRAY(n) OF <element\_type>;

**Using VARRAY in PL/SQL 👨‍💻**

Once a VARRAY type is defined, you can declare a variable of that type and initialize it using its constructor method, which has the same name as the VARRAY type. A VARRAY variable is NULL by default and must be initialized before you can reference its elements.

Example:

This program demonstrates how to declare a VARRAY type, create variables of that type, and iterate through them.

Code snippet

DECLARE

-- Define a VARRAY type for names

TYPE namesarray IS VARRAY(5) OF VARCHAR2(10);

-- Define a VARRAY type for marks

TYPE grades IS VARRAY(5) OF INTEGER;

-- Declare variables of the VARRAY types

names namesarray;

marks grades;

total INTEGER;

BEGIN

-- Initialize the VARRAYs using their constructors

names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');

marks := grades(98, 97, 78, 87, 92);

total := names.count; -- Get the number of elements

DBMS\_OUTPUT.PUT\_LINE('Total ' || total || ' Students');

FOR i IN 1 .. total LOOP

DBMS\_OUTPUT.PUT\_LINE('Student: ' || names(i) || ' Marks: ' || marks(i));

END LOOP;

END;

/

**Output:**

Total 5 Students

Student: Kavita Marks: 98

...

**VARRAY Attributes 🧩**

VARRAY elements can also be based on a database table's structure using the %TYPE attribute. This is useful for creating collections that mirror table columns.

Example:

This program uses a cursor to populate a VARRAY with customer names from a table, showcasing how to use %TYPE with a VARRAY.

Code snippet

DECLARE

-- Define a VARRAY type for customer names

TYPE c\_list IS VARRAY (6) OF customers.name%TYPE;

-- Declare a variable and initialize it

name\_list c\_list := c\_list();

-- Use a cursor to select names from the customers table

CURSOR c\_customers IS SELECT name FROM customers;

counter INTEGER := 0;

BEGIN

FOR n IN c\_customers LOOP

counter := counter + 1;

name\_list.EXTEND; -- Dynamically extend the array

name\_list(counter) := n.name;

DBMS\_OUTPUT.PUT\_LINE('Customer(' || counter || '): ' || name\_list(counter));

END LOOP;

END;

/

**PROCEDURES :**

A **subprogram** in PL/SQL is a named program unit that performs a specific task. They are a core component of modular programming, allowing for code reuse and better organization. PL/SQL offers two types of subprograms: **Procedures** and **Functions**. This chapter focuses on procedures.

**What is a PL/SQL Procedure? 🧑‍💻**

A procedure is a subprogram that primarily performs an action rather than returning a value. Procedures are stored in the database as schema objects and can be called from other PL/SQL blocks, subprograms, or applications.

A procedure has three parts, similar to a PL/SQL block:

* **Declarative Part**: Optional. Declares local variables, constants, cursors, and other subprograms. It does not start with the DECLARE keyword.
* **Executable Part**: Mandatory. Contains the statements that perform the procedure's logic. Enclosed between BEGIN and END.
* **Exception-handling Part**: Optional. Contains the code to handle runtime errors.

**Creating and Executing a Procedure ✅**

You create a standalone procedure using the CREATE OR REPLACE PROCEDURE statement.

**Syntax:**

Code snippet

CREATE [OR REPLACE] PROCEDURE procedure\_name [(parameter\_name [IN | OUT | IN OUT] type, ...)]

{IS | AS}

BEGIN

-- procedure\_body

END procedure\_name;

**Example:**

Code snippet

CREATE OR REPLACE PROCEDURE greetings AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

Once created, you can execute a standalone procedure in two ways:

1. Using the EXECUTE keyword: EXECUTE greetings;
2. Calling the procedure by name from another PL/SQL block: BEGIN greetings; END; /

**Parameter Modes in Procedures ⚙️**

Parameters allow a subprogram to accept and return values. PL/SQL has three parameter modes:

| Mode | Description | How It Works |
| --- | --- | --- |
| **IN** | Passes a value **into** the subprogram. It is a read-only parameter and is the **default** mode. | The value is passed from the caller to the subprogram. |
| **OUT** | Returns a value **out** from the subprogram to the caller. It acts like a variable inside the procedure; any value assigned to it is returned. | The value is passed from the subprogram back to the caller. |
| **IN OUT** | Passes a value **in** and returns a value **out**. The parameter can be read and modified within the subprogram. | The value is passed into the subprogram, modified, and then passed back out. |

**Example (IN and OUT):**

Code snippet

DECLARE

a NUMBER := 23;

b NUMBER := 45;

c NUMBER;

PROCEDURE findMin(x IN NUMBER, y IN NUMBER, z OUT NUMBER) IS

BEGIN

IF x < y THEN

z := x;

ELSE

z := y;

END IF;

END;

BEGIN

findMin(a, b, c);

dbms\_output.put\_line('Minimum of (' || a || ', ' || b || '): ' || c);

END;

/

**Methods for Passing Parameters ➡️**

Parameters can be passed to a procedure in three ways:

1. Positional Notation: Parameters are passed in the order they are declared in the procedure signature.

findMin(a, b, c);

1. Named Notation: Parameters are explicitly associated with their formal parameter names using =>. The order doesn't matter.

findMin(y => b, x => a, z => c);

1. Mixed Notation: A combination of both. Positional parameters must come before named parameters.

findMin(a, b, z => c);

**FUNCTIONS :**

A PL/SQL function is a named program unit similar to a procedure, but it **must return a value**. This key difference means that functions are primarily used to compute and return a single value, while procedures are used to perform an action.

**Creating and Calling a Function 📝**

You create a standalone function using the CREATE [OR REPLACE] FUNCTION statement. A function's syntax is similar to a procedure's, but it includes a mandatory RETURN clause that specifies the data type of the value it will return.

**Syntax:**

Code snippet

CREATE [OR REPLACE] FUNCTION function\_name

[(parameter\_name [IN | OUT | IN OUT] type, ...)]

RETURN return\_datatype

{IS | AS}

BEGIN

-- function\_body (must contain a RETURN statement)

END [function\_name];

To use a function, you call its name and pass any required parameters. The returned value can then be assigned to a variable.

Example:

This function, totalCustomers, returns the count of rows in the customers table.

Code snippet

CREATE OR REPLACE FUNCTION totalCustomers RETURN number IS

total NUMBER(2) := 0;

BEGIN

SELECT COUNT(\*) INTO total FROM customers;

RETURN total;

END;

/

You can call this function from another PL/SQL block:

Code snippet

DECLARE

c NUMBER(2);

BEGIN

c := totalCustomers();

DBMS\_OUTPUT.PUT\_LINE('Total no. of Customers: ' || c);

END;

/

**Output:**

Total no. of Customers: 6

**Recursive Functions ♻️**

A function is **recursive** when it calls itself within its own body. This is a powerful technique for solving problems that can be broken down into smaller, self-similar subproblems. A classic example is calculating a factorial.

Example (Factorial):

This function fact calculates the factorial of a number x by recursively calling itself with x-1 until x is 0.

Code snippet

DECLARE

num NUMBER := 6;

factorial NUMBER;

FUNCTION fact(x NUMBER) RETURN NUMBER IS

f NUMBER;

BEGIN

IF x = 0 THEN

f := 1;

ELSE

f := x \* fact(x - 1);

END IF;

RETURN f;

END;

BEGIN

factorial := fact(num);

DBMS\_OUTPUT.PUT\_LINE('Factorial of ' || num || ' is ' || factorial);

END;

/

**Output:**

Factorial of 6 is 720

**CURSORS :**

A **cursor** in PL/SQL is a pointer to a **context area** in memory created by Oracle to process a SQL statement. A cursor holds the set of rows (the **active set**) returned by a SQL query. They are used to fetch and process rows one at a time. There are two main types of cursors: **Implicit** and **Explicit**.

**Implicit Cursors 🕵️**

Implicit cursors are automatically created and managed by Oracle whenever a SQL statement is executed. Programmers cannot directly control them. For any INSERT, UPDATE, DELETE, or SELECT INTO statement, Oracle creates an implicit cursor. The most recent one is referred to as the **SQL cursor**.

You can access the attributes of the SQL cursor to check the outcome of a SQL statement. The most common attributes are:

| Attribute | Description |
| --- | --- |
| **SQL%FOUND** | Returns TRUE if a statement affected one or more rows. |
| **SQL%NOTFOUND** | Returns TRUE if a statement affected no rows. |
| **SQL%ISOPEN** | Always returns FALSE for an implicit cursor as it's closed immediately after execution. |
| **SQL%ROWCOUNT** | Returns the number of rows affected by the statement. |

Example:

This code updates salaries and checks how many rows were affected using SQL%ROWCOUNT.

Code snippet

DECLARE

total\_rows NUMBER(2);

BEGIN

UPDATE customers SET salary = salary + 500;

IF SQL%FOUND THEN

total\_rows := SQL%ROWCOUNT;

DBMS\_OUTPUT.PUT\_LINE(total\_rows || ' customers updated.');

END IF;

END;

/

**Explicit Cursors 🚶**

Explicit cursors are **user-defined** cursors that give you more control over the data retrieval process, particularly for SELECT statements that return multiple rows. An explicit cursor is declared in the DECLARE section of a PL/SQL block.

Working with an explicit cursor involves four steps:

1. **Declaration**: Define the cursor with a name and an associated SELECT statement.

Code snippet

CURSOR c\_customers IS SELECT id, name, address FROM customers;

1. **Opening**: The OPEN statement executes the SELECT query and loads the result set into the cursor's active set.

Code snippet

OPEN c\_customers;

1. **Fetching**: The FETCH statement retrieves one row at a time from the active set and assigns the column values to local variables. This is typically done within a loop.

Code snippet

FETCH c\_customers INTO c\_id, c\_name, c\_addr;

1. **Closing**: The CLOSE statement releases the memory allocated to the cursor, making it unusable until it is opened again.

Code snippet

CLOSE c\_customers;

**Example:**

Code snippet

DECLARE

c\_id customers.id%TYPE;

c\_name customers.name%TYPE;

c\_addr customers.address%TYPE;

CURSOR c\_customers IS SELECT id, name, address FROM customers;

BEGIN

OPEN c\_customers;

LOOP

FETCH c\_customers INTO c\_id, c\_name, c\_addr;

EXIT WHEN c\_customers%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(c\_id || ' ' || c\_name || ' ' || c\_addr);

END LOOP;

CLOSE c\_customers;

END;

/

**RECORDS :**

A **record** in PL/SQL is a composite data structure that can hold data items of different data types, similar to a row in a database table. Records are used to treat related data fields as a single logical unit, which makes code more readable and easier to manage.

**Types of Records 🗂️**

PL/SQL supports three types of records:

1. **Table-Based Records**: Created using the %ROWTYPE attribute on a database table. This automatically creates a record with fields that match the column names and data types of the specified table. This is the simplest way to fetch and manipulate an entire row of data.

**Example:**

Code snippet

DECLARE

customer\_rec customers%ROWTYPE; -- Declares a record matching the customers table

BEGIN

SELECT \* INTO customer\_rec

FROM customers

WHERE id = 5;

DBMS\_OUTPUT.PUT\_LINE('Customer Name: ' || customer\_rec.name);

END;

/

1. **Cursor-Based Records**: Created using the %ROWTYPE attribute on an explicit cursor. This is useful for fetching rows from a result set into a record, especially when the query involves joins or a subset of columns. The record's fields correspond to the columns in the cursor's SELECT list.

**Example:**

Code snippet

DECLARE

CURSOR customer\_cur IS SELECT id, name FROM customers;

customer\_rec customer\_cur%ROWTYPE; -- Declares a record for the cursor's result set

BEGIN

OPEN customer\_cur;

FETCH customer\_cur INTO customer\_rec;

-- ... further processing in a loop

CLOSE customer\_cur;

END;

/

1. **User-Defined Records**: You can create your own custom record structures by defining a TYPE with a list of fields and their data types. This allows you to group unrelated data items together into a logical unit.

**Syntax:**

Code snippet

TYPE type\_name IS RECORD (

field\_name1 datatype1,

field\_name2 datatype2,

...

);

record\_variable type\_name;

**Example:**

Code snippet

DECLARE

TYPE books IS RECORD (

title VARCHAR2(50),

author VARCHAR2(50),

book\_id NUMBER

);

book1 books; -- Declares a variable of the 'books' record type

BEGIN

book1.title := 'C Programming';

DBMS\_OUTPUT.PUT\_LINE('Book Title: ' || book1.title);

END;

/

You access individual fields of a record using the **dot (.) operator**.

**Records as Subprogram Parameters 🤝**

Records can be passed to and from procedures and functions just like any other variable. This simplifies passing multiple related values to a subprogram.

**Example:**

Code snippet

DECLARE

TYPE books IS RECORD (title VARCHAR2(50), ...);

book1 books;

PROCEDURE printbook(book books) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Book Title: ' || book.title);

END;

BEGIN

book1.title := 'C Programming';

printbook(book1); -- Pass the entire record as a parameter

END;

/

**EXCEPTIONS :**

An **exception** in PL/SQL is an error condition that occurs during program execution. PL/SQL provides a mechanism to handle these errors gracefully using an EXCEPTION block, preventing the program from terminating abruptly. There are two main types of exceptions: **system-defined** (predefined by Oracle) and **user-defined**.

**Exception Handling ⚙️**

The core of exception handling in PL/SQL is the EXCEPTION block, which is an optional section of a PL/SQL block. It contains handlers for specific exceptions. When an error occurs in the BEGIN...END block, program control is immediately transferred to the EXCEPTION block.

**Syntax:**

Code snippet

BEGIN

-- executable statements

EXCEPTION

WHEN exception1 THEN

-- handling for exception1

WHEN exception2 THEN

-- handling for exception2

...

WHEN others THEN

-- catch-all for any other exceptions

END;

* **WHEN others**: This is a catch-all handler that traps any exception not explicitly handled by a preceding WHEN clause. It is a good practice to include it to ensure your program doesn't fail unexpectedly.

Example:

The SELECT INTO statement in this example fails because no customer with ID = 8 exists, which raises the system-defined exception NO\_DATA\_FOUND. The EXCEPTION block catches this and prints a user-friendly message.

Code snippet

DECLARE

c\_id customers.id%type := 8;

c\_name customers.name%type;

c\_addr customers.address%type;

BEGIN

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

EXCEPTION

WHEN no\_data\_found THEN

DBMS\_OUTPUT.PUT\_LINE('No such customer!');

WHEN others THEN

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred!');

END;

/

**User-Defined Exceptions 🛠️**

Programmers can create their own exceptions to handle specific error conditions not covered by Oracle's predefined exceptions.

1. **Declaration**: A user-defined exception must be declared in the DECLARE section.

Code snippet

DECLARE

my\_exception EXCEPTION;

1. **Raising**: It must be explicitly raised using the RAISE statement.

Code snippet

IF condition THEN

RAISE my\_exception;

END IF;

Example:

This program raises a custom exception ex\_invalid\_id if the user enters a non-positive customer ID.

Code snippet

DECLARE

c\_id customers.id%type := &cc\_id;

ex\_invalid\_id EXCEPTION; -- Declare user-defined exception

BEGIN

IF c\_id <= 0 THEN

RAISE ex\_invalid\_id; -- Raise the custom exception

ELSE

-- ...

END IF;

EXCEPTION

WHEN ex\_invalid\_id THEN

DBMS\_OUTPUT.PUT\_LINE('ID must be greater than zero!');

END;

/

**Pre-defined Exceptions 🚨**

PL/SQL has many built-in exceptions that are raised automatically when a specific database error occurs. Some of the most common ones include:

* **NO\_DATA\_FOUND**: Raised when a SELECT INTO statement returns zero rows.
* **TOO\_MANY\_ROWS**: Raised when a SELECT INTO statement returns more than one row.
* **DUP\_VAL\_ON\_INDEX**: Raised when you try to insert a duplicate value into a column that has a unique index.
* **ZERO\_DIVIDE**: Raised when you attempt to divide a number by zero.
* **INVALID\_NUMBER**: Raised when the conversion of a character string to a number fails.

|  |  |  |  |
| --- | --- | --- | --- |
| **Exception** | **Oracle Error** | **SQLCODE** | **Description** |
| ACCESS\_INTO\_NULL | 06530 | -6530 | It is raised when a null object is automatically assigned a value. |
| CASE\_NOT\_FOUND | 06592 | -6592 | It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION\_IS\_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP\_VAL\_ON\_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |
| INVALID\_CURSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID\_NUMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN\_DENIED | 01017 | -1017 | It is raised when a program attempts to log on to the database with an invalid username or password. |
| NO\_DATA\_FOUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT\_LOGGED\_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM\_ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE\_MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |
| SELF\_IS\_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE\_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO\_MANY\_ROWS | 01422 | -1422 | It is raised when a SELECT INTO statement returns more than one row. |
| VALUE\_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |