

PL/SQL

- Procedural Language extensions to SQL.
- Developed by Oracle for its relational database system
- Combines the power of SQL with procedural programming features (control flow, loops, etc.)

Why use PL/SQL?

- Enhanced data manipulation capabilities beyond basic SQL
- Improved performance for complex operations
- Reusability through procedures, functions, and packages
- Encapsulation of business logic within the database

Key Features:

- **Procedural constructs:** control flow statements (if-then-else, loops) for complex logic
- **Cursors:** for iterating through result sets of SQL statements
- **Error handling:** exceptions to manage errors during program execution
- **Integration with SQL:** execute SQL statements directly within PL/SQL code

Example - Simple PL/SQL Block:

```
DECLARE
    bonus NUMBER := 100; -- Variable declaration
BEGIN
    -- Update employee table with bonus amount
    UPDATE employees
    SET salary = salary + bonus
    WHERE department_id = 10;

    DBMS_OUTPUT.PUT_LINE('Salary updated for department 10
employees!');
END;
/
```

Advantages of PL/SQL

- **Combines SQL and Procedural Code:** Write logic and manipulate data within the same program.
- **Improved Performance:** Compiled code reduces network traffic and improves speed.
- **Increased Productivity:** Functions, procedures, and loops enable modular and reusable code.
- **Portable and Scalable:** Works across Oracle environments and handles complex tasks efficiently.
- **Object-Oriented Features:** Promotes better code organization.
- **Error Handling and Security:** Manages errors and restricts unauthorized access.
- **Web Application Development:** Can be used to create dynamic web content.

PL/SQL Blocks

- **Purpose:** Encapsulate a set of PL/SQL statements to perform a specific task.
- **Structure:**
 - **DECLARE:** Define variables, cursors, exceptions, etc. (Optional)
 - **BEGIN:** Marks the start of executable statements.
 - **Executable Statements:** Code to perform operations like calculations, data manipulation, control flow.
 - **END:** Marks the end of the block.
- **Example:**

```

DEDECLARE
    bonus NUMBER(10,2);
BEGIN
    bonus := salary * 0.1; -- Calculate bonus (assuming salary
variable exists)
    INSERT INTO bonuses (employee_id, amount) VALUES (emp_id,
bonus);
END;
/

```

Character Set

- **Definition:** Set of characters a database can understand and store.
- **Importance:** Determines valid characters in identifiers, literals, and comments.

- Unicode character sets like UTF-8 are commonly used for internationalization and multilingual support.
- **Common Character Sets:** AL32UTF8, WE8ISO8859P1 (depends on your database setup)
- **Impact on PL/SQL:**
 - Influences what characters you can use in variable names, string literals, etc.
 - Affects how data is interpreted if transferred between databases with different character sets.

Literals

- **Definition:** Fixed values directly included in your code.
- **Types:**
 - Numeric (e.g., 123, 3.14)
 - Character (e.g., 'A', 'Hello World')
 - String literals (enclosed in single quotes, can include multiple characters)
 - Boolean (TRUE, FALSE, NULL)

Additional Notes:

- PL/SQL is case-insensitive except for characters within single quotes (string literals).
- Use appropriate data types for literals to ensure data integrity.

PL/SQL Data Types

1. Scalar Data Types: Represent single values.

- **NUMBER:** Stores numeric values with precision and scale (e.g., `salary NUMBER(8,2)`)
- **BOOLEAN:** Represents true or false (e.g., `is_active BOOLEAN`)
- **CHAR/VARCHAR2:** Stores fixed or variable length character strings (e.g., `name VARCHAR2(50)`)
- **DATE:** Stores date and time information (e.g., `hire_date DATE`)

Subtypes:

- Derived from base types (like NUMBER) with additional constraints.
- Examples:
 - **POSITIVE:** Restricts a number to positive values (e.g., `age POSITIVE`)
 - **NATURAL:** Restricts a number to non-negative values (e.g., `balance NATURAL`)

2. Collections: Group similar data items together.

- **VARRAY:** Ordered collection with a fixed size at declaration (e.g., `emp_ids VARRAY(100) OF NUMBER`)
- **TABLE:** Associative collection with key-value pairs (e.g., ``customer_data TABLE OF VARCHAR2(20)`)

3. Large Objects (LOBs): Store large data like text, images, or audio.

- **BLOB:** Binary Large Object (e.g., `product_image BLOB`)
- **CLOB:** Character Large Object (e.g., `product_description CLOB`)

4. Record Types: Group related variables of different data types into a single unit.

5. Reference Types:

- Store references to existing data structures.
- Example: `emp_cursor REF CURSOR;` (reference to a cursor)

Variables

- In PL/SQL, variables act as temporary storage locations for data during program execution.
- They allow you to manipulate data within your code.

Declaring Variables:

- Variables need to be declared before use within a PL/SQL block using the **DECLARE** keyword.
- Declaration specifies the variable name, data type, and optionally an initial value.



```
DECLARE
    v_emp_id NUMBER(4) := 100;  -- Variable to store employee ID
                                (number)
    v_name VARCHAR2(50) := 'John Doe';  -- Variable to store
    employee name (string)
BEGIN
    -- Code to use v_emp_id and v_name variables
END;
/
```

Constants

Constants are fixed values that cannot be changed during program execution. Think of them as named literals.

Why use them?

- **Improve code readability:** Use meaningful names instead of hardcoded values.
- **Easier maintenance:** Change the value in one place (constant declaration) instead of searching throughout the code.
- **Enforce data type:** Ensures the constant value is compatible with its intended use.

Declaring Constants: using the **CONSTANT** keyword.

```
CONSTANT constant_name data_type := value [NOT NULL];
```

Points to Remember:

- Constant values cannot be modified after declaration.
- Use appropriate data types to avoid implicit conversions.
- Constants improve code maintainability and readability.

Attributes in PL/SQL

There are two main contexts for attributes in PL/SQL:

- **Cursor Attributes:** These attributes provide information about the state of a cursor.
- **%TYPE Attribute:** This attribute helps in declaring variables based on existing data types.

1. Cursor Attributes:

- Provide information on a cursor's status after execution.
- Common cursor attributes include:
 - **%ISOPEN:** Checks if the cursor is open.
 - **%FOUND:** Indicates if a FETCH retrieved a row (TRUE) or not (FALSE).
 - **%NOTFOUND:** Opposite of %FOUND (TRUE if no row fetched, FALSE otherwise).
 - **%ROWCOUNT:** Returns the number of rows fetched using the cursor.

2. %TYPE Attribute:

- Used in variable declarations to reference data types of existing columns or variables.

- Ensures type compatibility between the PL/SQL variable and the referenced source.
- Syntax: `<variable_name> %TYPE (<schema>.<table_name>.<column_name> | <previously_declared_variable>)`

Control Structures

PL/SQL offers three primary control structures to manage the flow of execution in your code:

1. Conditional Control (Selection Structures)

- Determines which block of code to execute based on a boolean condition (TRUE or FALSE).

IF-THEN-ELSE Statement:

```
IF condition1 THEN
    -- code to execute if condition1 is TRUE
ELSIF condition2 THEN
    -- code to execute if condition1 is FALSE and condition2 is TRUE
ELSE
    -- code to execute if all conditions are FALSE
END IF;
```

CASE Statement:


- Evaluates an expression against multiple conditions and executes corresponding code blocks.
- Example: Assigning a letter grade based on a score

```
CASE expression
WHEN value1 THEN
    -- statements to execute if expression evaluates to value1
WHEN value2 THEN
    -- statements to execute if expression evaluates to value2
ELSE
    -- statements to execute if none of the WHEN conditions match
END CASE;
```

2. Iterative Control (Looping Structures)

- Executes a block of code repeatedly until a specific condition is met.


LOOP Statement:



```
LOOP
    statements;
    EXIT WHEN condition;  -- Optional exit condition
END LOOP;
```

FOR Loop:


- Iterates a specific number of times based on a counter variable.
- Example: Calculating the factorial of a number



```
FOR loop_counter IN lower_bound .. upper_bound LOOP
    -- statements to execute for each iteration
END LOOP;
```

WHILE Loop:

- Continues execution as long as a condition remains TRUE.
- Example: Reading employee records until a specific ID is found



```
WHILE condition LOOP
    -- statements to execute as long as condition is TRUE
END LOOP;
```

Sequential Control (limited use in PL/SQL)

- Executes statements in the order they appear in your code.

- **GOTO:** (Generally discouraged due to potential for making code harder to read and maintain)

Key Points:

- Conditional control allows for different code paths based on conditions.
- Iterative control enables repeated execution until a stopping condition is reached.
- Sequential control is the default flow of execution.

Cursors in PL/SQL

Concept:

- A cursor acts as a pointer to a result set retrieved from a SELECT statement.
- It allows you to process data one row at a time, unlike fetching the entire set upfront.

Types of Cursors:

1. Implicit Cursors:

- Created automatically by PL/SQL for any SELECT statement without an explicitly declared cursor.
- Offer limited control.

2. Explicit Cursors:

- Defined by the programmer for granular control over data retrieval.
- Involve declaring, opening, fetching, and closing the cursor.

Steps to Use Explicit Cursors (Example):

Declare the Cursor:

```
CURSOR cursor_name IS  
    SELECT statement;
```

Open the Cursor:



```
OPEN cursor_name;
```

Fetch Data (Loop):



```
LOOP  
  FETCH cursor_name INTO variable_list;  
  EXIT WHEN cursor_name%NOTFOUND;  
  -- Process data in the variable_list  
END LOOP;
```

Close the Cursor:



```
CLOSE cursor_name;
```

Benefits of Explicit Cursors:

- Enhanced control over data processing.
- Memory efficiency when dealing with large datasets.

Considerations:

- Use cursors judiciously, as frequent cursor usage can impact performance.
- For simple row fetching, consider alternatives like BULK COLLECT or FORALL loops.

Exception Handling

Exceptions: Errors that disrupt normal program flow during execution.

Two Types:

- **System-defined Exceptions:** Predefined errors by PL/SQL (e.g., `ZERO_DIVIDE`, `NO_DATA_FOUND`).

- **User-defined Exceptions:** Programmer-created exceptions for specific error conditions (e.g., `INVALID_INPUT`).

Key Concepts:

- **RAISE Statement:** Explicitly raises an exception to signal an error.
 - Example: `RAISE NO_DATA_FOUND;` (when a query retrieves no rows)
- **Exception Block:** Captures and handles raised exceptions.
 - Follows the `BEGIN` block.

```
EXCEPTION
  WHEN exception_name THEN
    -- Handler code for specific exception
  WHEN OTHERS THEN -- Optional, handles any uncaught exception
    -- Generic handler code
END;
```

Benefits:

- **Robust Code:** Handles errors gracefully, preventing program crashes.
- **Improved Readability:** Separates error handling from core logic.
- **Maintainability:** Easier to identify and fix errors.

Triggers

Stored procedures: Like stored procedures, triggers are PL/SQL blocks stored in the database and can be invoked repeatedly. However, unlike procedures, you cannot explicitly call triggers; they are automatically executed upon specific events.

Event-driven execution: Triggers fire in response to various database events, including:

- **DML (Data Manipulation Language) events:** These involve modifications to tables or views using `INSERT`, `UPDATE`, or `DELETE` statements.
- **DDL (Data Definition Language) events:** These involve schema changes using `CREATE`, `ALTER`, or `DROP` statements.
- **Database events:** These encompass system-level events like server errors, user logins/logouts, database startup/shutdown.

Trigger Timing:

- **BEFORE:** Executes the trigger code before the triggering event (INSERT, UPDATE, DELETE) occurs. Useful for data validation or setting derived column values.
- **AFTER:** Executes the trigger code after the triggering event has been applied. Useful for auditing or performing actions based on changes.
- **INSTEAD OF:** Less common, replaces the standard DML operation with custom logic defined in the trigger.

Trigger Benefits:

- **Enforce data integrity:** Triggers can validate data before inserts or updates, preventing invalid entries.
- **Automate tasks:** Automate tasks like calculating derived columns, logging changes, or enforcing referential integrity.
- **Implement security:** Triggers can restrict data access based on user roles or implement additional security checks.
- **Data synchronization:** Keep data consistent across multiple tables by replicating changes in triggers.

Procedures

Subprograms: Procedures are reusable blocks of PL/SQL code that perform specific tasks. They promote modular design by encapsulating functionality.

Structure: A procedure consists of a header (specification) and a body:

- **Header:** Defines the procedure name, optional parameters (IN for input, OUT for output), and return type (if applicable).
- **Body:** Contains the executable code (declarations, statements, exception handling).

Benefits:

- **Reusability:** Procedures can be called from other PL/SQL code, reducing redundancy.
- **Modularity:** Break down complex tasks into smaller, manageable units.
- **Maintainability:** Easier to modify and debug code within procedures.
- **Performance:** Can improve performance by reducing network traffic compared to multiple single SQL statements.

Packages

What are they?

- A way to organize and group related PL/SQL code elements.
- Think of them as reusable modules containing procedures, functions, variables, cursors, and more.

Benefits:

- **Modularity:** Break down large codebases into smaller, manageable units.
- **Reusability:** Share common code across different parts of your application.
- **Maintainability:** Easier to understand, modify, and debug code.
- **Encapsulation:** Control access to code elements (public vs. private).

Package Specification:

- Declares the public interface (available elements).
- Includes declarations for procedures, functions, variables, cursors, etc.

```
PACKAGE calculate_shipping IS
    PROCEDURE calculate_cost(weight NUMBER, distance NUMBER) PUBLIC;
END PACKAGE calculate_shipping;
```

Package Body (Optional):

- Contains the implementation (code) for public elements.
- Also includes private declarations for helper functions or variables.

Using Packages:

- Reference public elements using the `package_name.element_name` syntax.

```
BEGIN
    calculate_shipping.calculate_cost(10, 200);
END;
```