Database development with PL/SQL INSY 8311

ADVENTIST UNIVERSITY OF CENTRAL AFRICA

Instructor:

- Eric Maniraguha | ericmaniraguha2024@gmail.com | LinkedIn Profile

6h00 pm – 8h50 pm

- Monday A -G209
- Tuesday B-G209
- Wednesday E-G209
- Thursday F-G308

September 2024

Database development with PL/SQL

Reference reading

- PL/SQL Tutorial

Lecture 06 – PL/SQL Programming Fundamentals

PL/SQL Block Structure

A PL/SQL block is the fundamental unit of a PL/SQL program. It consists of three main sections: **the declaration section**, **the begin section, and the exception handling section**. Each section serves a specific purpose, and together they allow you to write organized and efficient code.

```
DECLARE
   -- Declaration section: Declare variables, constants, types, and exceptions
BEGIN
   -- Execution section: Write the executable statements
EXCEPTION
   -- Exception handling section: Handle errors and exceptions
END;
```

PL/SQL Block Structure

A PL/SQL block can consist of up to four different sections, with only one section being mandatory:

1. Header
- **Description**: Used only for named blocks. It defines how the block or program can be called.
- **Optional**: This section is not required for anonymous blocks.

2. Declaration Section
- **Description**: This section is where you declare variables, cursors, and sub-blocks that will be used in the execution and exception sections.
- **Optional**: This section can be omitted if there are no variables or cursors to declare.

3. Execution Section
- **Description**: Contains the statements that the PL/SQL runtime engine will execute during runtime.
- **Mandatory**: This section is required in every PL/SQL block.

4. Exception Section
- **Description**: This section is used to handle exceptions that occur during the execution of the block, such as warnings and error conditions.
- **Optional**: You can choose to include this section if you want to manage exceptions, but it is not required.


```
PROCEDURE get_happy (ename_in IN VARCHAR2)      <-- Header
IS
   l_hiredate DATE;                             <-- Declaration
BEGIN
   l_hiredate := SYSDATE - 2;
   INSERT INTO employee                         <-- Execution
      (emp_name, hiredate)
   VALUES (ename_in, l_hiredate);
EXCEPTION
   WHEN DUP_VAL_IN_INDEX
   THEN
      DBMS_OUTPUT.PUT_LINE                      <-- Exception
         ('Cannot insert.');
END;
```

Source Image: A procedure containing all four sections

Summary
In summary, while the execution section is essential for any PL/SQL block, the header, declaration, and exception sections are optional and can be included based on the needs of your program.

The 'Hello World' Example

DECLARE
message varchar2(20):= 'Hello, World!';
BEGIN
dbms_output.put_line(message);
END; /
	Output:
	
	Hello World
	
	PL/SQL procedure successfully completed.

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result –

PL/SQL Identifiers

In PL/SQL, identifiers refer to names used for various program elements, including **constants, variables, exceptions, procedures, cursors, and reserved words**. Here are the key features of PL/SQL identifiers:

- **Character Composition:** An identifier must start with a letter and can be followed by letters, numerals, dollar signs **($),** underscores **(_),** and number signs **(#).**
- **Length:** Identifiers cannot exceed **30 characters**.
- **Case Sensitivity:** By default, identifiers are not case-sensitive. For example, **number** and **NUMBER** are treated the same.
- **Reserved Words:** You cannot use reserved keywords (like **SELECT, BEGIN**, etc.) as identifiers.

Example of Valid Identifiers:

- **employee_name**
- **salary$amount**
- **total_count_1**

Example of Invalid Identifiers:

- **1st_employee** (cannot start with a numeral)
- **select** (reserved word)

PL/SQL Delimiters: symbol with a special meaning

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

Delimiter	Description
;	Statement terminator
:=	Assignment operator
=>	Association operator
`	
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin/end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, =, !=, ^=	Different versions of NOT EQUAL

SQL*Plus

Conclusion

Understanding PL/SQL identifiers and delimiters is essential for writing clear and effective code. Proper naming conventions for identifiers help improve code readability, while knowledge of delimiters ensures that the code structure is maintained correctly. For those looking to deepen their SQL skills, consider enrolling in an SQL certification course to advance your career with real-world projects.

Operators and Expressions in PL/SQL

1. **Arithmetic Operators: Perform mathematical operations.**
   - + : Addition
   - - : Subtraction
   - * : Multiplication
   - / : Division
   - MOD : Modulus (remainder of division)

```
DECLARE
   a NUMBER := 10;
   b NUMBER := 3;
   c NUMBER;
   rounded_c NUMBER;
BEGIN
   c := a + b; -- Addition
   dbms_output.put_line('Sum: ' || c);

   c := a / b; -- Division without rounding
   dbms_output.put_line('Division (not rounded): ' || c);

   rounded_c := ROUND(a / b, 2); -- Division with rounding to 2 decimal places
   dbms_output.put_line('Division (rounded): ' || rounded_c);

   c := a MOD b; -- Modulus
   dbms_output.put_line('Remainder: ' || c);
END;
/
```


```
Sum: 13
Division (not rounded): 3.3333333333333333
Division (rounded): 3.33
Remainder: 1


PL/SQL procedure successfully completed.
```

Relational (Comparison) Operators: Compare two values and return a boolean (TRUE, FALSE, or NULL).
   - = : Equal to
   - != or <> : Not equal to
   - > : Greater than
   - < : Less than
   - >= : Greater than or equal to
   - <= : Less than or equal to

```
DECLARE
   a NUMBER := 10;
   b NUMBER := 20;
BEGIN
   IF a = b THEN
      dbms_output.put_line('a is equal to b');
   ELSE
      dbms_output.put_line('a is not equal to b');
   END IF;
END;
/
```

```
a is not equal to b


PL/SQL procedure successfully completed.
```

Operators and Expressions in PL/SQL

Concatenation Operator: Used to combine two strings.
▪ **||: Concatenates two strings.**
	DECLARE
	first_name VARCHAR2(20) := Didier';
	last_name VARCHAR2(20) := 'Ndayisenga';
	BEGIN
	dbms_output.put_line('Full Name: ' || first_name || ' ' || last_name);
	END;
	/
		Full Name: Didier Ndayisenga
		 
		
		PL/SQL procedure successfully completed.

Unary Operators: Operators that work with a single operand.
▪ +: Indicates a positive number.
▪ -: Negates a numbe
	DECLARE
	a NUMBER := -10;
	BEGIN
	dbms_output.put_line('Negative number: ' || a);
	END;
	/
		Negative number: -10
		PL/SQL procedure successfully completed.

Assignment Operator: Used to assign a value to a variable.
:=: Assigns a value to a variable.
	DECLARE
	a NUMBER;
	BEGIN
	a := 100; -- Assigning 100 to variable a
	dbms_output.put_line('Value of a: ' || a);
	END;
	/
		DECLARE
		a NUMBER;
		BEGIN
		a := 100; -- Assigning 100 to variable a
		dbms_output.put_line('Value of a: ' || a);
		END;
		/

Operators and Expressions in PL/SQL

2. Expressions in PL/SQL
 	
An expression is a combination of one or more values, variables, constants, and operators that are evaluated to produce a result. The result of an expression can be of any data type.
▪ Simple Expression: Direct assignment or operation between two variables or constants.
	DECLARE
	  a NUMBER := 10;
	  b NUMBER := 20;
	  result NUMBER;
	BEGIN
	  result := a + b; -- Simple arithmetic expression
	  dbms_output.put_line('Result: ' || result);
	END;
	/
		Result: 30
		
		PL/SQL procedure successfully completed.

Conditional Expression: Using relational and logical operators to check conditions.	DECLARE
	  age NUMBER := 18;
	BEGIN
	  IF age >= 18 THEN
	  dbms_output.put_line('You are eligible to vote.');
	  ELSE
	  dbms_output.put_line('You are not eligible to vote.');
	  END IF;
	END;
	/
		You are eligible to vote.
		
		PL/SQL procedure successfully completed.

String Expression: Concatenating strings using the || operator.	DECLARE
	  first_name VARCHAR2(20) := 'John';
	  last_name VARCHAR2(20) := 'Doe';
	BEGIN
	  dbms_output.put_line('Full Name: ' || first_name || ' ' || last_name);
	END;
	/
		Full Name: John Doe
		
		PL/SQL procedure successfully completed.

Types of Comments

1. Single-Line Comments

- Single-line comments in PL/SQL start with -- (double hyphen).
- Everything after -- on the same line is considered a comment and will be ignored by the compiler.

Example:

```
DECLARE
   -- This is a single-line comment
   message  VARCHAR2(20) := 'Hello, World!';
BEGIN
   DBMS_OUTPUT.PUT_LINE(message); -- Display the message
END;
/
```

2. Multi-Line Comments

Multi-line comments start with **/* and end with */.**
This type of comment can span multiple lines, and everything between **/* and */** will be treated as a comment.

Example:

```
DECLARE
   message  VARCHAR2(20) := 'Hello, World!';
BEGIN
   /* This is a multi-line comment
      explaining the logic inside
      the PL/SQL block. */
   DBMS_OUTPUT.PUT_LINE(message);
END;
/
```

Using Comments in PL/SQL

- Single-line comments are useful for quick explanations of a particular statement or variable declaration.
- Multi-line comments are ideal for providing detailed explanations or documentation for larger sections of code.

Example:

```
DECLARE
   -- Declaring a variable to hold the message
   message  VARCHAR2(20) := 'Hello, World!';
BEGIN
   /*
      This block will print the message
      to the output console.
   */
   DBMS_OUTPUT.PUT_LINE(message); -- Display the message
END;
/
```

Conclusion

Using comments in your PL/SQL code is a good practice that helps in maintaining and understanding the code more effectively. Whether you are writing code for yourself or for others, comments play an important role in explaining the purpose, logic, and functionality of the code.

Data Type

A **data type** in programming defines the kind of data a variable can hold and how that data can be used or manipulated. In PL/SQL (and other programming languages), data types help the system understand the format, range, and behavior of the data. This ensures that the operations performed on data are valid and meaningful.

Oracle SQL datatype (1/2)

SQL*Plus

Character Datatypes

- **CHAR(size)**: Fixed-length character data (size bytes). Default and minimum size are 1 byte.
- **VARCHAR2(size)**: Variable-length character data. Maximum size is 4000 bytes.
- **NCHAR(size)**: Fixed-length Unicode character data (size characters).
- **NVARCHAR2(size)**: Variable-length Unicode character data. Maximum size is 4000 bytes.
- **CLOB**: Character Large Object, used for storing large blocks of text up to 4 GB.
- **NCLOB**: Unicode CLOB, used for large blocks of multi-byte character data.

Numeric Datatypes

- **NUMBER(p, s)**: Variable precision and scale numeric data. Precision (p) can range from 1 to 38, and scale (s) can range from -84 to 127.
- **BINARY_FLOAT**: 32-bit single-precision floating-point number.
- **BINARY_DOUBLE**: 64-bit double-precision floating-point number.

Date and Time Datatypes

- **DATE**: Stores date and time information to the second, ranging from January 1, 4712 BC, to December 31, 9999 AD.
- **TIMESTAMP [(fractional_seconds_precision)]**: Date and time with fractional seconds precision. Precision can range from 0 to 9.
- **TIMESTAMP WITH TIME ZONE**: Includes time zone displacement.
- **TIMESTAMP WITH LOCAL TIME ZONE**: Normalizes time to the database time zone.
- **INTERVAL YEAR TO MONTH**: Stores a period of time in years and months.
- **INTERVAL DAY TO SECOND**: Stores a period of time in days, hours, minutes, and seconds.

Oracle SQL datatype (2/2)

LOB (Large Object) Datatypes

- **BLOB**: Binary Large Object, used for storing large binary data (e.g., images, videos).
- **BFILE**: Binary File, used to store a pointer to an external file.

Raw and Rowid Datatypes

- **RAW(size)**: Stores binary data of variable length, up to 2000 bytes.
- **LONG RAW**: Stores variable-length binary data up to 2 GB (deprecated, use BLOB instead).
- **ROWID**: Stores a unique identifier for rows in a table.

XML and JSON Datatypes

- **XMLTYPE**: Used to store and manipulate XML data.
- **JSON**: Used to store JSON (JavaScript Object Notation) data.

Spatial and Multimedia Datatypes

- **SDO_GEOMETRY**: Used for storing spatial data (geographical data).
- **ORDIMAGE**: Used for storing image data.

These datatypes allow Oracle SQL to handle a wide range of data efficiently, providing powerful capabilities for data storage, retrieval, and manipulation.

Importance of Datatype in Oracle SQL (1/2)

Data Integrity and Accuracy

- **Purpose**: Datatypes ensure that the data stored in a database is of the correct type and format. For example, a column defined as NUMBER can only store numeric values, which prevents errors like storing text in a numerical field.
- **Benefit**: This helps maintain the accuracy and consistency of the data, reducing the chances of data corruption and logical errors.

Memory and Storage Optimization

- **Purpose**: Different datatypes consume different amounts of storage space. For instance, a CHAR(10) column reserves 10 bytes for each entry, while a VARCHAR2(10) only uses the space needed for the actual characters stored.
- **Benefit**: Choosing the appropriate datatype optimizes disk space usage and improves performance, especially in large databases.

Performance Optimization

- **Purpose**: Datatypes influence how data is indexed, searched, and processed. Numeric types are faster for calculations than strings, and properly indexed date types can speed up time-based queries.
- **Benefit**: Using the right datatype enhances query performance, speeds up data retrieval, and reduces processing time.

Validation and Constraints

- **Purpose**: Datatypes enforce basic validation rules automatically. For example, an attempt to insert a non-date value into a DATE column will result in an error.
- **Benefit**: This reduces the need for additional validation checks in the application layer, simplifying code and improving reliability.

Importance of datatype in Oracle SQL (2/2)

Data Type-Specific Functions

- **Purpose**: Certain operations and functions are designed to work with specific datatypes, like AVG() for numbers or TO_DATE() for date strings.
- **Benefit**: Proper datatype usage allows you to leverage these functions effectively, making data manipulation and analysis easier and more efficient.

Security and Protection

- **Purpose**: Certain datatypes, like BLOB or CLOB, help protect sensitive data by ensuring that data is stored and handled correctly, especially when dealing with large or binary files.
- **Benefit**: Datatype constraints prevent data misuse or misinterpretation, enhancing overall data security.

Consistency Across Applications

- **Purpose**: Datatypes help ensure consistent data representation across different applications interacting with the database.
- **Benefit**: Consistency minimizes integration issues and helps maintain data uniformity when used by different systems or platforms.

Summary

- Datatypes are not just about categorizing data; they are critical for maintaining data quality, optimizing storage and performance, ensuring consistency, and enabling advanced data manipulation capabilities within databases

Example of Data Type

```
SET SERVEROUTPUT ON;

DECLARE
    v_name VARCHAR2(50);  -- String data type
    v_age NUMBER(3);      -- Numeric data type
    v_birthdate DATE;     -- Date data type
    v_is_active BOOLEAN;  -- Boolean data type
BEGIN
    v_name := 'Alice';
    v_age := 25;
    v_birthdate := TO_DATE('1999-01-01', 'YYYY-MM-DD');
    v_is_active := TRUE;

    DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Age: ' || v_age);
    DBMS_OUTPUT.PUT_LINE('Birthdate: ' || v_birthdate);

    -- Use an IF statement to handle BOOLEAN output
    IF v_is_active THEN
        DBMS_OUTPUT.PUT_LINE('Active: TRUE');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Active: FALSE');
    END IF;
END;
```

Name: Alice
Age: 25
Birthdate: 01-JAN-99
Active: TRUE

PL/SQL procedure successfully completed.

Explanation:

The || operator is used for concatenating strings, but BOOLEAN values cannot be directly concatenated. So instead, I've used an IF statement to check the value of **v_is_active** and print either 'TRUE' or 'FALSE' accordingly.

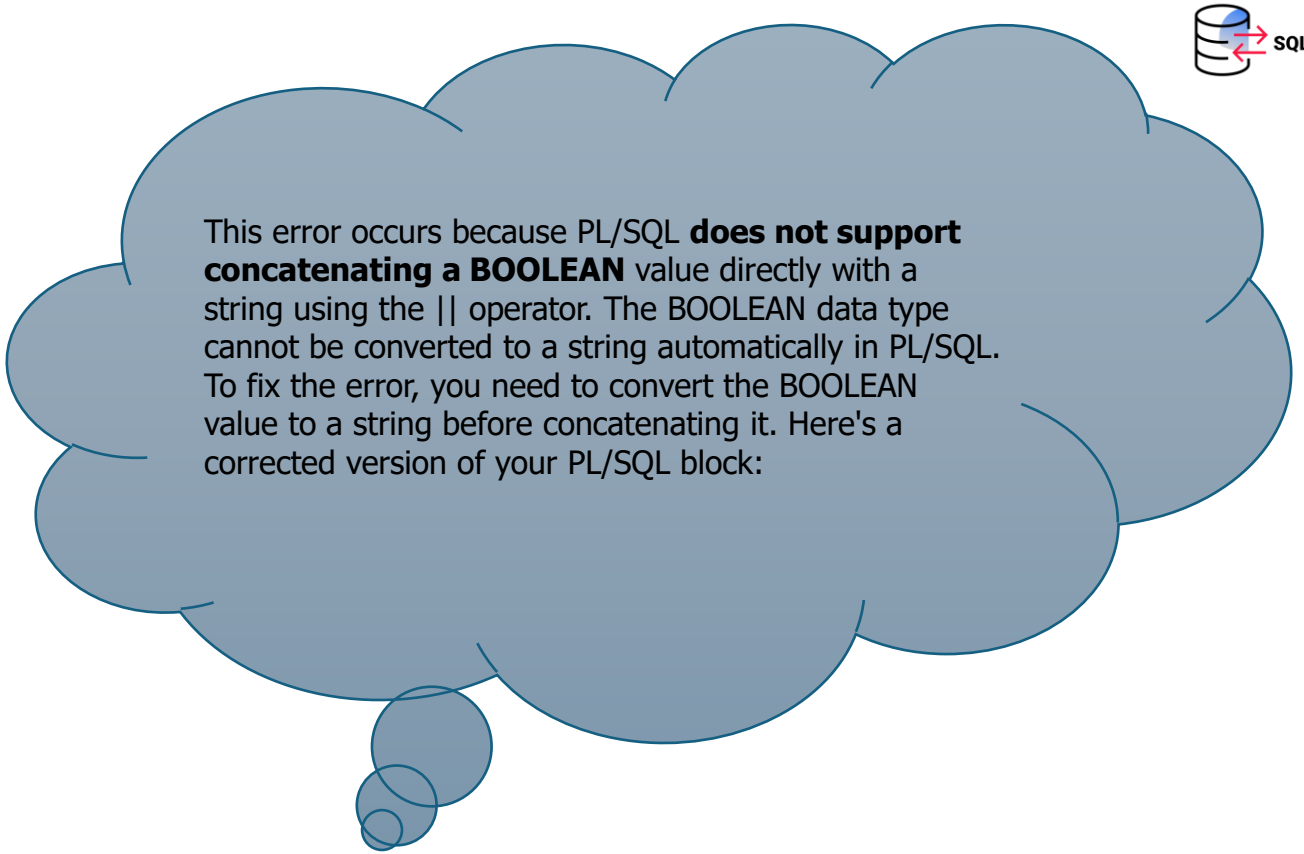
Common PL/SQL Errors: Concatenation of BOOLEAN with Strings

```
SET SERVEROUTPUT ON;

DECLARE
    v_name VARCHAR2(50);  -- String data type
    v_age NUMBER(3);      -- Numeric data type
    v_birthdate DATE;     -- Date data type
    v_is_active BOOLEAN;  -- Boolean data type
BEGIN
    v_name := 'Alice';
    v_age := 25;
    v_birthdate := TO_DATE('1999-01-01', 'YYYY-MM-DD');
    v_is_active := TRUE;

    DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Age: ' || v_age);
    DBMS_OUTPUT.PUT_LINE('Birthdate: ' || v_birthdate);
    DBMS_OUTPUT.PUT_LINE('Active: ' || v_is_active);
END;
```

```
Error report -
ORA-06550: line 15, column 26:
PLS-00306: wrong number or types of arguments in call to '||'
ORA-06550: line 15, column 5:
PL/SQL: Statement ignored
06550. 00000 -  "line %s, column %s:\n%s"
*Cause:    Usually a PL/SQL compilation error.
*Action:
```



Working with Date Formats in PL/SQL

Example: Displaying Dates in Different Formats

```
SELECT
    SYSDATE AS "Default Format",
    TO_CHAR(SYSDATE, 'YYYY-MM-DD') AS "ISO Format",
    TO_CHAR(SYSDATE, 'DD-MON-YYYY') AS "Day-Month-Year Format",
    TO_CHAR(SYSDATE, 'DD Month YYYY') AS "Full Month Name Format",
    TO_CHAR(SYSDATE, 'MM/DD/YYYY') AS "US Format",
    TO_CHAR(SYSDATE, 'Day, DDth Month YYYY') AS "Custom Format",
    TO_CHAR(SYSDATE, 'YYYY/MM/DD HH24:MI:SS') AS "Timestamp Format"
FROM dual;
```

Explanation:

- SYSDATE returns the current system date and time in the default format (DD-MON-YY or similar).
- TO_CHAR(SYSDATE, 'YYYY-MM-DD') converts the date to the YYYY-MM-DD ISO format.
- TO_CHAR(SYSDATE, 'DD-MON-YYYY') shows the date with the day, abbreviated month, and full year.
- TO_CHAR(SYSDATE, 'DD Month YYYY') shows the date with the full month name.
- TO_CHAR(SYSDATE, 'MM/DD/YYYY') formats the date in the traditional US format.
- TO_CHAR(SYSDATE, 'Day, DDth Month YYYY') includes the full day name and ordinal day (e.g., 01st, 02nd).
- TO_CHAR(SYSDATE, 'YYYY/MM/DD HH24:MI:SS') shows the full timestamp with a 24-hour clock.

Example Output:

Default Format	ISO Format	Day-Month-Year Format	Full Month Name Format	US Format	Custom Format	Timestamp Format
20-SEP-24	2024-09-20	20-SEP-2024	20 September 2024	09/20/2024	Friday, 20th September 2024	2024/09/20 14:35:22

PL/SQL Variables: Declaration, Data Types, and Usage

Variables in PL/SQL are used to store data temporarily in memory while the program is running. They allow you to manipulate and process information, such as storing results from a query, performing calculations, or holding values that change during execution.

Key Points About Variables:

- **Declaration**: You must declare a variable before using it. This involves specifying the variable's name and its data type.
- **Assignment**: Variables can be assigned values using the **:= operator.**
- **Scope**: A variable's scope determines where it can be accessed, typically within the block in which it is declared.

Variable Declaration Syntax:

variable_name data_type [NOT NULL] := initial_value;

- **variable_name:** The name of the variable.
- **data_type:** The type of data the variable can store (e.g., VARCHAR2, NUMBER, DATE).
- **NOT NULL:** Optional. Enforces that the variable cannot store a NULL value.
- **initial_value:** Optional. The starting value assigned to the variable.

Key Takeaways:

- **Variables** are essential for storing and manipulating data in PL/SQL programs.
- You must declare variables before using them.
- **:=** is used for assigning values to variables.
- The **DBMS_OUTPUT.PUT_LINE** procedure helps in displaying variable values, but you need to convert certain types like DATE and handle types like BOOLEAN appropriately.

Initializing Variables in PL/SQL

When you declare a variable in PL/SQL, **it is automatically assigned a default value of NULL.** If you wish to initialize a variable with a specific value at the time of declaration, you can do so in one of two ways:

1.Using the DEFAULT Keyword
2.Using the Assignment Operator

counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';

Additionally, you can specify that a variable should not hold a ***NULL*** value by using the ***NOT NULL*** constraint. When you use this constraint, you must provide an explicit initial value for that variable.
Properly initializing variables is considered good programming practice, as failing to do so can lead to unexpected results.

Example of Variable Initialization:

```
DECLARE
   -- Declare two integer variables a and b with initial values
   a integer := 10;
   b integer := 20;
   -- Declare an integer variable c to store the sum of a and b
   c integer;
   -- Declare a real variable f to store the result of a floating-point division
   f real;
BEGIN
   -- Calculate the sum of a and b, and assign it to c
   c := a + b;
   -- Output the value of c to the console
   dbms_output.put_line('Value of c: ' || c);
   -- Perform floating-point division and assign the result to f
   f := 70.0 / 3.0;
   -- Output the value of f to the console
   dbms_output.put_line('Value of f: ' || f);
END;
/
```

Execution Result:

```
Value of c: 30
Value of f: 23.333333333333333

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

PL/SQL supports nested blocks, meaning each program block can contain inner blocks. A variable declared within an inner block is not accessible to the outer block. However, a variable declared in the outer block can be accessed by all nested inner blocks. There are two types of variable scope:

- **Local Variables**: Declared in an inner block and not accessible to outer blocks.
- **Global Variables**: Declared in the outermost block or a package.

Example of Variable Scope:

```
DECLARE
   -- Global variables
   num1 number := 95;
   num2 number := 85;

BEGIN
   dbms_output.put_line('Outer Variable num1: ' || num1);
   dbms_output.put_line('Outer Variable num2: ' || num2);

   DECLARE
      -- Local variables
      num1 number := 195;
      num2 number := 185;

   BEGIN
      dbms_output.put_line('Inner Variable num1: ' || num1);
      dbms_output.put_line('Inner Variable num2: ' || num2);
   END;
END;
/
```

Execution Result:

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

Conclusion

Understanding how to properly initialize variables and the concept of variable scope in PL/SQL is crucial for effective programming. By following best practices in variable management, you can avoid potential pitfalls and ensure that your programs produce the expected results. For those looking to deepen their SQL knowledge, consider enrolling in an SQL certification course to enhance your skills and advance your career.

Variable Initialization - Exercise

Exercise 1

Objective: Practice declaring and initializing variables with different data types.

1. Task: Write a PL/SQL block that declares the following variables:

- student_name (VARCHAR2) and initialize it with your name.
- student_id (NUMBER) and initialize it with your ID number.
- enrollment_date (DATE) and initialize it with the current date.
- is_enrolled (BOOLEAN) and initialize it to TRUE.

2. Output: Use DBMS_OUTPUT.PUT_LINE to display each variable's value in the following format:

Student Name: [name]
Student ID: [ID]
Enrollment Date: [date]
Is Enrolled: [true/false]

3. Example Output:

Student Name: Alice
Student ID: 12345
Enrollment Date: 20-SEP-2024
Is Enrolled: TRUE

Challenge yourself!

Calculations with Variables - Exercise

Exercise 2:

Objective: Practice performing arithmetic operations and displaying results.

1. Task: Write a PL/SQL block that declares and initializes the following variables:

- math_score (NUMBER) with a value of 85.
- science_score (NUMBER) with a value of 90.
- english_score (NUMBER) with a value of 88.
- average_score (NUMBER) (do not initialize this variable yet).

2. Calculate: Compute the average score by summing math_score, science_score, and english_score, and then dividing by 3.

3. Output: Display the average score using DBMS_OUTPUT.PUT_LINE.

4. Example Output:

Average Score: 87.67

Bonus Challenge: Combine Exercises

1.Task:

Create a comprehensive PL/SQL block that combines aspects from the above exercises. Include variable initialization, arithmetic calculations, handling NULL values, and demonstrate local and global variable scope.

Tips for Students:

- Always enable DBMS_OUTPUT to see the output of your program.
- Use comments in your code to explain each section.
- Test your code for different scenarios to understand variable behavior fully.
- These exercises will help students solidify their understanding of PL/SQL variables, their initialization, and their scope.

Key Points and Benefit of Constants

Key Points:

- **Immutability**: Once a constant is declared and assigned a value, it cannot be modified during the execution of the block.
- **Performance**: Using constants can improve performance, as their values are set once and referenced without the overhead of variable reassignment.
- **Maintainability**: Constants help make code more maintainable by ensuring that important values (such as mathematical constants, limits, or thresholds) remain consistent throughout the program.

Benefits of Using Constants:

- **Avoid Errors**: Ensures that important values cannot be accidentally modified.
- **Code Readability**: Improves readability by giving descriptive names to fixed values.
- **Consistency**: Promotes consistency across the code by using the same constant value in multiple places without the risk of mismatch.
- **Maintainability**: If the constant value needs to be updated (e.g., business rule change), it only needs to be updated in one place, making the code easier to maintain.

Constants are especially useful when dealing with values that should not change, such as configuration parameters, mathematical values, or application limits.

Constants in PL/SQL

A **constant** in PL/SQL is a variable whose value is assigned once and cannot be changed during the execution of the program. Declaring a constant ensures that its value remains fixed throughout the block or procedure, making the code more reliable and easier to maintain.

How to Declare Constants

To declare a constant in PL/SQL:

- Use the keyword CONSTANT in the variable declaration.
- Assign a value at the time of declaration using the **:= operator.**
- Once assigned, the value of the constant cannot be changed.

Syntax for Declaring a Constant:

constant_name CONSTANT datatype := value;

- **constant_name:** The name of the constant.
- **CONSTANT:** The keyword indicating that this variable is a constant.
- **datatype:** The datatype of the constant (e.g., **NUMBER, VARCHAR2**, etc.).
- **value:** The value assigned to the constant, which cannot be modified later.

```
DECLARE
   -- Declaring a constant for the value of Pi
   pi CONSTANT NUMBER := 3.14159;
   -- Declaring a constant for maximum number of employees
   max_employees CONSTANT INTEGER := 100;
BEGIN
   -- Trying to change the value of pi would result in an error:
   -- pi := 3.14; -- This would cause a compilation error.

   dbms_output.put_line('Value of Pi: ' || pi);
   dbms_output.put_line('Maximum Employees: ' || max_employees);
END;
/
```

What is PL/SQL Program Units

A **PL/SQL program unit** refers to any block of PL/SQL code that can be compiled, stored, and executed in a database. PL/SQL program units include a variety of structures used for defining, organizing, and reusing code. Each of these units plays a different role in the programming and functioning of database applications.

PL/SQL Block: Basic unit of PL/SQL code with optional declaration and exception handling.

Function: Subprogram that returns a value.

Package: A group of related functions, procedures, and types.

Package Body: Implementation of the functions and procedures in the package.

Procedure: Subprogram that performs a task without returning a value.

Trigger: Automatically executed code in response to specific database events.

Type: Defines custom data structures (e.g., objects or records).

Type Body: Implements methods for object types.

Keep Learning!