3

Writing Executable Statements

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- Identify lexical units in a PL/SQL block
- Use built-in SQL functions in PL/SQL
- Describe when implicit conversions take place and when explicit conversions have to be dealt with
- Write nested blocks and qualify variables with labels
- Write readable code with appropriate indentations

Lesson Aim

You have learned how to declare variables and write executable statements in a PL/SQL block. In this lesson, you learn how lexical units make up a PL/SQL block. You learn to write nested blocks. You also learn about the scope and visibility of variables in the nested blocks and about qualifying them with labels.

Lexical Units in a PL/SQL Block

Lexical units:

- **Are building blocks of any PL/SQL block**
- **Are sequences of characters including letters, numerals, tabs, spaces, returns, and symbols**
- **Can be classified as:**
  - **Identifiers**
  - **Delimiters**
  - **Literals**
  - **Comments**

Lexical Units in a PL/SQL Block

Lexical units include letters, numerals, special characters, tabs, spaces, returns, and symbols.

- **Identifiers: Identifiers are the names given to PL/SQL objects. You have learned to identify valid and invalid identifiers. Recall that keywords cannot be used as identifiers.**

  Quoted Identifiers:

  - **Make identifiers case sensitive**
  - **Include characters such as spaces**
  - **Use reserved words**

  Examples:

  ```
  "begin date" DATE;

  "end date"   DATE;
  "exception thrown" BOOLEAN DEFAULT TRUE;
  ```

  All subsequent usage of these variables should have double quotation marks.

- **Delimiters: Delimiters are symbols that have special meaning. You have**

Lexical Units in a PL/SQL Block (continued)

- **Delimiters (continued)**
  Delimiters are simple or compound symbols that have special meaning in PL/SQL.

Simple Symbols:

Symbol	Meaning
+	Addition operator
-	Subtraction/negation operator
*	Multiplication operator
/	Division operator
=	Equality operator
@	Remote access indicator
;	Statement terminator

Compound Symbols

Symbol	Meaning
<>	Inequality operator
!=	Inequality operator
||	Concatenation operator
--	Single-line comment indicator
/*	Beginning comment delimiter
*/	Ending comment delimiter
:=	Assignment operator

Note: This is only a subset and not a complete list of delimiters.

- **Literals: Any value that is assigned to a variable is a literal. Any character, numeral, Boolean, or date value that is not an identifier is a literal. Literals are classified as:**
  - **Character literals: All string literals have the data type `CHAR` and are therefore called character literals (for example, `John`, `12C`, `1234`, and `12-JAN-1923`).**
  - **Numeric literals: A numeric literal represents an integer or real value (for example, `428` and `1.276`).**
  - **Boolean literals: Values that are assigned to Boolean variables are Boolean literals. `TRUE`, `FALSE`, and `NULL` are Boolean literals or keywords.**
- **Comments: It is good programming practice to explain what a piece of code is trying to achieve. When you include the explanation in a PL/SQL block, the compiler cannot interpret these instructions. There should be a way in which you can indicate that these instructions need not be compiled. Comments are mainly used for this purpose. Any instruction that is commented is not interpreted by the compiler.**

PL/SQL Block Syntax and Guidelines

- **Literals:**
  - **Character and date literals must be enclosed in single quotation marks.**

```
name := 'Henderson';
```

  - **Numbers can be simple values or scientific notation.**
- **Statements can continue over several lines.**

PL/SQL Block Syntax and Guidelines

A literal is an explicit numeric, character string, date, or Boolean value that is not represented by an identifier.

- **Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols.**
- **Numeric literals can be represented either by a simple value (for example, –32.5) or in scientific notation (for example, 2E5 means $2 * 10^5 = 200,000$).**

Commenting Code

- Prefix single-line comments with two hyphens (--).
- Place multiple-line comments between the symbols /* and */.

Example

```
DECLARE
...
annual_sal NUMBER (9,2);
BEGIN    -- Begin the executable section

/* Compute the annual salary based on the
   monthly salary input from the user */
annual_sal := monthly_sal * 12;
END;     -- This is the end of the block
/
```

ORACLE

Commenting Code

You should comment code to document each phase and to assist debugging. Comment the PL/SQL code with two hyphens (--) if the comment is on a single line, or enclose the comment between the symbols /* and */ if the comment spans several lines.

Comments are strictly informational and do not enforce any conditions or behavior on logic or data. Well-placed comments are extremely valuable for code readability and future code maintenance. In the example in the slide, the lines enclosed within /* and */ indicate a comment that explains the following code.

SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row number
  - Single-row character
  - Data type conversion
  - Date
  - Timestamp
  - `GREATEST` and `LEAST`
  - Miscellaneous functions
- Not available in procedural statements:
  - `DECODE`
  - Group functions

SQL Functions in PL/SQL

SQL provides a number of predefined functions that can be used in SQL statements. Most of these functions are valid in PL/SQL expressions.

The following functions are not available in procedural statements:

- `DECODE`
- Group functions: `AVG`, `MIN`, `MAX`, `COUNT`, `SUM`, `STDDEV`, and `VARIANCE`
  Group functions apply to groups of rows in a table and therefore are available only in SQL statements in a PL/SQL block.

The functions mentioned here are only a subset of the complete list.

SQL Functions in PL/SQL: Examples

- Get the length of a string:

```
desc_size INTEGER(5);
prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

-- get the length of the string in prod_description
desc_size:= LENGTH(prod_description);
```

- Convert the employee name to lowercase:

```
emp_name:= LOWER(emp_name);
```

ORACLE

SQL Functions in PL/SQL: Examples

SQL functions help you to manipulate data. They are grouped into the following categories:

- Number
- Character
- Conversion
- Date
- Miscellaneous

Data Type Conversion

- Convert data to comparable data types
- Are of two types:
  - Implicit conversions
  - Explicit conversions
- Some conversion functions:
  - `TO_CHAR`
  - `TO_DATE`
  - `TO_NUMBER`
  - `TO_TIMESTAMP`

Data Type Conversion

In any programming language, converting one data type to another is a common requirement. PL/SQL can handle such conversions with scalar data types. Data type conversions can be of two types:

Implicit conversions: PL/SQL attempts to convert data types dynamically if they are mixed in a statement. Consider the following example:

```
DECLARE
 salary NUMBER(6):=6000;
 sal_hike VARCHAR2(5):='1000';
 total_salary salary%TYPE;
BEGIN
 total_salary:=salary+sal_hike;
END;
/
```

In the example shown, the variable `sal_hike` is of type `VARCHAR2`. While calculating the total salary, PL/SQL first converts `sal_hike` to `NUMBER` and then performs the operation. The result is of the `NUMBER` type.

Implicit conversions can be between:

Data Type Conversion (continued)

Explicit conversions: To convert values from one data type to another, use built-in functions. For example, to convert a `CHAR` value to a `DATE` or `NUMBER` value, use `TO_DATE` or `TO_NUMBER`, respectively.

Data Type Conversion

1
```
date_of_joining DATE:= '02-Feb-2000';
```

2
```
date_of_joining DATE:= 'February 02,2000';
```

3
```
date_of_joining DATE:= TO_DATE('February
02,2000','Month DD, YYYY');
```

ORACLE

Data Type Conversion (continued)

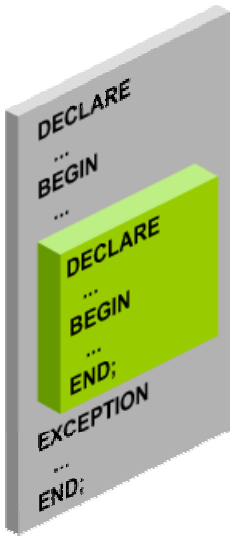
Implicit and explicit conversions of the `DATE` data type:

1. **This example of implicit conversion assigns the date `date_of_joining`.**
2. **PL/SQL gives you an error because the date that is being assigned is not in the default format.**
3. **Use the `TO_DATE` function to explicitly convert the given date in a particular format and assign it to the `DATE` data type variable `date_of_joining`.**

Nested Blocks

PL/SQL blocks can be nested.

- An executable section (`BEGIN ... END`) can contain nested blocks.
- An exception section can contain nested blocks.



ORACLE

Nested Blocks

One of the advantages of PL/SQL (compared to SQL) is the ability to nest statements.
You can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. If your executable section has code for many logically related functionalities to support multiple business requirements, you can divide the executable section into smaller blocks. The exception section can also contain nested blocks.

Nested Blocks

Example

```
DECLARE
 outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
   inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
   DBMS_OUTPUT.PUT_LINE(inner_variable);
   DBMS_OUTPUT.PUT_LINE(outer_variable);
  END;
 DBMS_OUTPUT.PUT_LINE(outer_variable);
END;
/
```

ORACLE

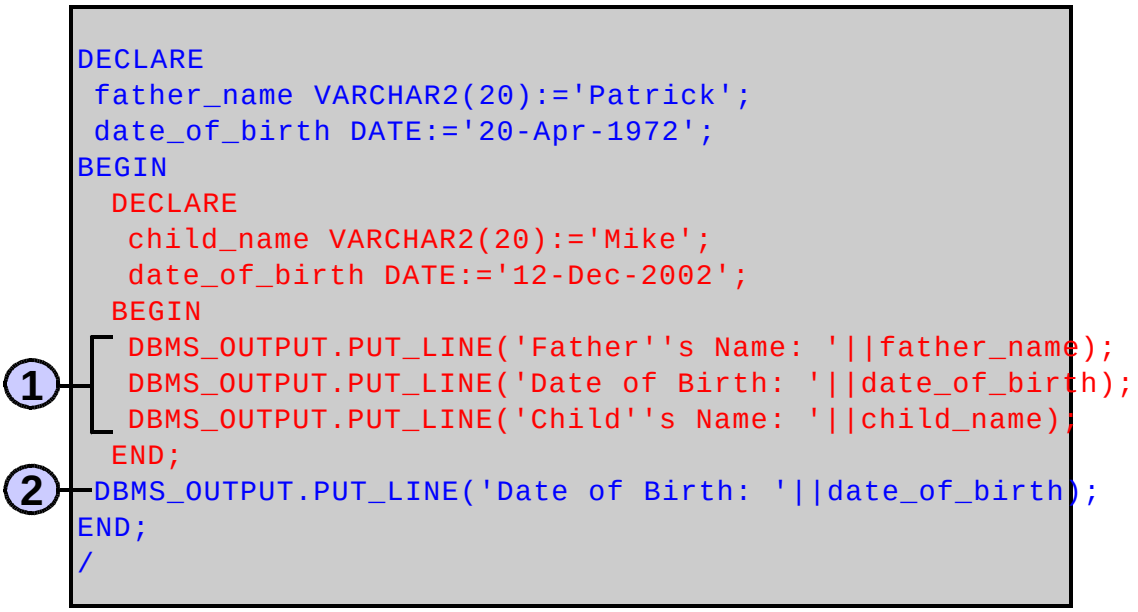
Nested Blocks (continued)

The example shown in the slide has an outer (parent) block and a nested (child) block. The variable `outer_variable` is declared in the outer block and the variable `inner_variable` is declared in the inner block.

`outer_variable` is local to the outer block but global to the inner block. When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name. There is no variable with the same name in the inner block, so PL/SQL looks for the variable in the outer block. Therefore, `outer_variable` is considered the global variable for all the enclosing blocks. You can access this variable in the inner block as shown in the slide. Variables declared in a PL/SQL block are considered local to that block and global to all its subblocks.

The `inner_variable` variable is local to the inner block and is not global because the inner block does not have any nested blocks. This variable can be accessed only within the inner block. If PL/SQL does not find the variable declared locally, it looks upward in the declarative section of the parent blocks. PL/SQL does not look downward in the child blocks.

Variable Scope and Visibility



Variable Scope and Visibility

The output of the block shown in the slide is as follows:

```
Father's Name: Patrick
Date of Birth: 12-DEC-02
Child's Name: Mike
Date of Birth: 20-APR-72
PL/SQL procedure successfully completed.
```

Examine the date of birth that is printed for father and child.

The *scope* of a variable is the portion of the program in which the variable is declared and is accessible.

The *visibility* of a variable is the portion of the program where the variable can be accessed without using a qualifier.

Scope

- **The variables `father_name` and `date_of_birth` are declared in the outer block. These variables have the scope of the block in which they are declared and accessible. Therefore, the scope of these variables is limited to the outer block.**

Variable Scope and Visibility (continued)

Scope (continued)

- The variables `child_name` and `date_of_birth` are declared in the inner block or the nested block. These variables are accessible only within the nested block and are not accessible in the outer block. When a variable is out of scope, PL/SQL frees the memory used to store the variable; therefore, these variables cannot be referenced.

Visibility

- The `date_of_birth` variable declared in the outer block has the scope even in the inner block. However, this variable is not visible in the inner block because the inner block has a local variable with the same name.

1. Examine the code in the executable section of the PL/SQL block. You can print the father's name, the child's name, and the date of birth. Only the child's date of birth can be printed here because the father's date of birth is not visible.
2. The father's date of birth is visible here and therefore can be printed.

You cannot have variables with the same name in a block. However, you can declare variables with the same name in two different blocks (nested blocks). The two items represented by the identifiers are distinct; changes in one do not affect the other.

Qualify an Identifier

```
<<outer>>
DECLARE
 father_name VARCHAR2(20):='Patrick';
 date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
   child_name VARCHAR2(20):='Mike';
   date_of_birth DATE:='12-Dec-2002';
  BEGIN
   DBMS_OUTPUT.PUT_LINE('Father''s Name: '||father_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '
                        ||outer.date_of_birth);
   DBMS_OUTPUT.PUT_LINE('Child''s Name: '||child_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '||date_of_birth);
  END;
END;
/`
```

ORACLE

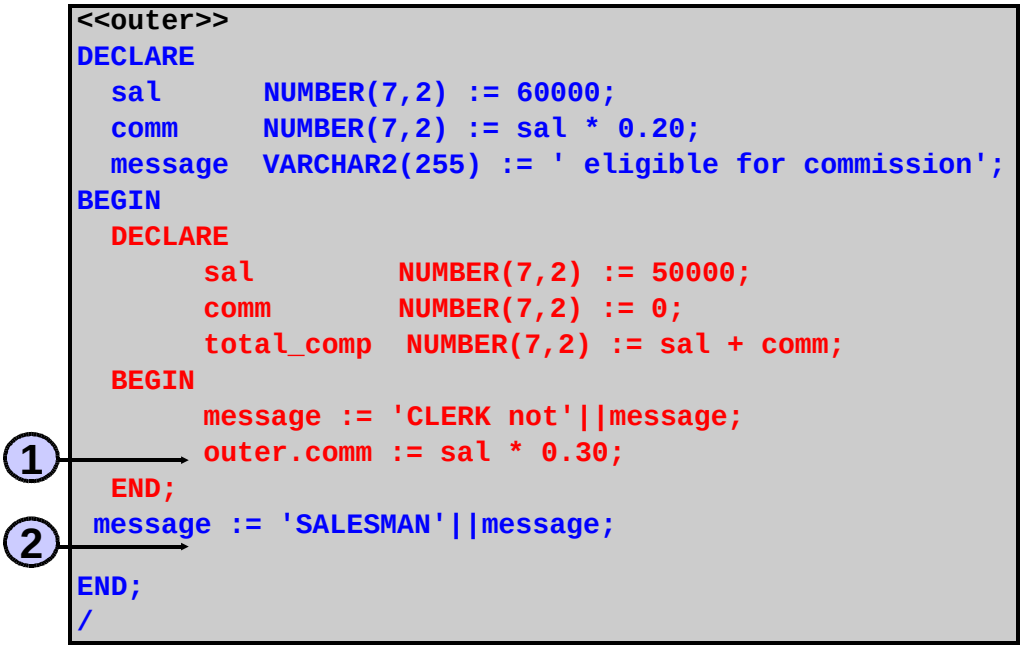
Qualify an Identifier

A qualifier is a label given to a block. You can use a qualifier to access the variables that have scope but are not visible. Examine the code: You can now print the father's date of birth and the child's date of birth in the inner block. The outer block is labeled `outer`. You can use this label to access the `date_of_birth` variable declared in the outer block.

Because labeling is not limited to the outer block, you can label any block. following:

```
Father's Name: Patrick
Date of Birth: 20-APR-72
Child's Name: Mike
Date of Birth: 12-DEC-02
PL/SQL procedure successfully completed.
```

Determining Variable Scope



ORACLE

Determining Variable Scope

Evaluate the PL/SQL block in the slide. Determine each of the following values according to the rules of scoping:

1. **Value of `MESSAGE` at position 1**
2. **Value of `TOTAL_COMP` at position 2**
3. **Value of `COMM` at position 1**
4. **Value of `outer.COMM` at position 1**
5. **Value of `COMM` at position 2**
6. **Value of `MESSAGE` at position 2**

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations

} Same as in SQL

- Exponential operator (**)

Operators in PL/SQL

The operations in an expression are performed in a particular order depending on their precedence (priority). The following table shows the default order of operations from high priority to low priority:

Operator	Operation
**	Exponentiation
+, -	Identity, negation
*, /	Multiplication, division
+, -, ||	Addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation
AND	Conjunction
OR	Inclusion

Operators in PL/SQL

Examples

- **Increment the counter for a loop.**

```
loop_count := loop_count + 1;
```

- **Set the value of a Boolean flag.**

```
good_sal := sal BETWEEN 50000 AND 150000;
```

- **Validate whether an employee number contains a value.**

```
valid  := (empno IS NOT NULL);
```

ORACLE

Operators in PL/SQL (continued)

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- **Comparisons involving nulls always yield `NULL`.**
- **Applying the logical operator `NOT` to a null yields `NULL`.**
- **In conditional control statements, if the condition yields `NULL`, its associated sequence of statements is not executed.**

Programming Guidelines

Make code maintenance easier by:

- **Documenting code with comments**
- **Developing a case convention for the code**
- **Developing naming conventions for identifiers and other objects**
- **Enhancing readability by indenting**

Programming Guidelines

Follow programming guidelines shown in the slide to produce clear code and reduce maintenance when developing a PL/SQL block.

Code Conventions

The following table provides guidelines for writing code in uppercase or lowercase characters to help distinguish keywords from named objects.

Category	Case Convention	Examples
SQL statements	Uppercase	`SELECT, INSERT`
PL/SQL keywords	Uppercase	`DECLARE, BEGIN, IF`
Data types	Uppercase	`VARCHAR2, BOOLEAN`
Identifiers and parameters	Lowercase	`v_sal, emp_cursor, g_sal, p_empno`
Database tables and columns	Lowercase	`employees, employee_id, department_id`

Indenting Code

For clarity, indent each level of code.

Example:

```
BEGIN
  IF x=0 THEN
     y:=1;
  END IF;
END;
/
```

```
DECLARE
  deptno       NUMBER(4);
  location_id  NUMBER(4);
BEGIN
  SELECT  department_id,
          location_id
  INTO    deptno,
          location_id
  FROM    departments
  WHERE   department_name
          = 'Sales';
...
END;
/
```

ORACLE

Indenting Code

For clarity and enhanced readability, indent each level of code. To show structure, you can divide lines by using carriage returns and you can indent lines by using spaces and tabs. Compare the following `IF` statements for readability:

```
IF x>y THEN max:=x;ELSE max:=y;END IF;
```

```
IF x > y THEN
  max := x;
ELSE
  max := y;
END IF;
```

Summary

In this lesson, you should have learned how to:

- Use built-in SQL functions in PL/SQL
- Write nested blocks to break logically related functionalities
- Decide when to perform explicit conversions
- Qualify variables in nested blocks

Summary

Because PL/SQL is an extension of SQL, the general syntax rules that apply to SQL also apply to PL/SQL.

A block can have any number of nested blocks defined within its executable part. Blocks defined within a block are called subblocks. You can nest blocks only in the executable part of a block. Because the exception section is also in the executable section, you can have nested blocks in that section. Ensure correct scope and visibility of the variables when you have nested blocks. Avoid using the same identifiers in the parent and child blocks.

Most of the functions available in SQL are also valid in PL/SQL expressions. Conversion functions convert a value from one data type to another. Comparison operators compare one expression to another. The result is always `TRUE`, `FALSE`, or `NULL`. Typically, you use comparison operators in conditional control statements and in the `WHERE` clause of SQL data manipulation statements. The relational operators enable you to compare arbitrarily complex expressions.

Practice 3: Overview

This practice covers the following topics:

- **Reviewing scoping and nesting rules**
- **Writing and testing PL/SQL blocks**

ORACLE

Practice 3: Overview

Exercises 1 and 2 are paper based.

Practice 3

Note: It is recommended to use *i*SQL*Plus for this practice.

PL/SQL Block

```
DECLARE
 weight    NUMBER(3) := 600;
 message   VARCHAR2(255) := 'Product 10012';
BEGIN
  DECLARE
   weight   NUMBER(3) := 1;
   message   VARCHAR2(255) := 'Product 11001';
   new_locn  VARCHAR2(50) := 'Europe';
  BEGIN
1  weight := weight + 1;
   new_locn := 'Western ' || new_locn;

  END;
 weight := weight + 1;
2 message := message || ' is in stock';
 new_locn := 'Western ' || new_locn;

END;
/
```

1. **Evaluate the PL/SQL block given above and determine the data type and value of each of the following variables according to the rules of scoping.**
   a. **The value of `weight` at position 1 is:**

   b. **The value of `new_locn` at position 1 is:**

   c. **The value of `weight` at position 2 is:**

   d. **The value of `message` at position 2 is:**

   e. **The value of `new_locn` at position 2 is:**

Practice 3 (continued)

Scope Example

```
DECLARE
  customer      VARCHAR2(50) := 'Womansport';
  credit_rating   VARCHAR2(50) := 'EXCELLENT';
BEGIN
   DECLARE
     customer        NUMBER(7) := 201;
     name     VARCHAR2(25) := 'Unisports';
   BEGIN
     credit_rating :='GOOD';
     …
   END;
  …
END;
/
```

2. **In the PL/SQL block shown above, determine the values and data types for each of the following cases.**

   a. **The value of `customer` in the nested block is:**

   b. **The value of `name` in the nested block is:**

   c. **The value of `credit_rating` in the nested block is:**

   d. **The value of `customer` in the main block is:**

   e. **The value of `name` in the main block is:**

   f. **The value of `credit_rating` in the main block is:**

Practice 3 (continued)

3. **Use the same session that you used to execute the practices in Lesson 2. If you have opened a new session, then execute `lab_02_05_soln.sql`. Edit `lab_02_05_soln.sql`.**

   - **Use single line comment syntax to comment the lines that create the bind variables.**
   - **Use multiple line comments in the executable section to comment the lines that assign values to the bind variables.**
   - **Declare two variables: `fname` of type `VARCHAR2` and size 15, and `emp_sal` of type `NUMBER` and size 10.**
   - **Include the following SQL statement in the executable section:**

     ```
     SELECT first_name, salary
         INTO fname, emp_sal FROM employees
         WHERE employee_id=110;
     ```

   - **Change the line that prints 'Hello World' to print 'Hello' and the first name. You can comment the lines that display the dates and print the bind variables, if you want to.**
   - **Calculate the contribution of the employee towards provident fund (PF).**
     PF is 12% of the basic salary and basic salary is 45% of the salary.
     Hello John
     YOUR SALARY IS : 8200
     YOUR CONTRIBUTION TOWARDS PF: 442.8
     PL/SQL procedure successfully completed.
     and use only one
     loyee's salary and his
   - **Execute and save your script as `lab_03_03_soln.sql`. Sample output is shown below.**

4. **Accept a value at run time using the substitution variable. In this practice, you will modify the script that you created in exercise 3 to accept user input.**

   - **Load the script `lab_03_04.sql` file.**
   - **Include the `PROMPT` command to prompt the user with the following message:**
     'Please enter your employee number.'
   - **Modify the declaration of the `empno` variable to accept the user input.**
   - **Modify the select statement to include the variable `empno`**

Practice 3 (continued)

Enter 100 and click the Continue button.

```
Hello Steven
YOUR SALARY IS : 24000
YOUR CONTRIBUTION TOWARDS PF: 1296
PL/SQL procedure successfully completed.
```

5. **Execute the script `lab_03_05.sql`. This script creates a table called `employee_details`.**
   a. **The `employee` and `employee_details` tables have the same data. You will update the data in the `employee_details` table. Do not update or change the data in the `employees` table.**
   b. **Open the script `lab_03_05b.sql` and observe the code in the file. Note that the code accepts the employee number and the department number from the user.**
   c. **You will use this as the skeleton script to develop the application, which was discussed in the lesson titled "Introduction."**