Executable Statements

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 indentation
- Use sequences in PL/SQL expressions

You 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 nested blocks and about qualifying variables with labels.

Lexical Units in a PL/SQL Block

- Are building blocks of any PL/SQL block
- Lexical units include letters, numerals, special characters, tabs, spaces, returns, and symbols.
- Can be classified as:
  - **Identifiers:** Identifiers are the names given to PL/SQL objects. `v_fname, c_percent`
  - **Delimiters:** Delimiters are symbols that have special meaning`; , +, -`
  - **Literals:** Any value that is assigned to a variable is a literal. `John, 428, True`
  - **Comments:** `--, /* */`

- **Identifiers:** Identifiers are the names given to PL/SQL objects. You 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. However, use of quoted identifiers is not recommended.
- **Delimiters:** Delimiters are symbols that have special meaning. You already learned that the semicolon (`;`) is used to terminate a SQL or PL/SQL statement. Therefore, `;` is an example of a delimiter.
  For more information, refer to the *PL/SQL User's Guide and Reference*.

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.

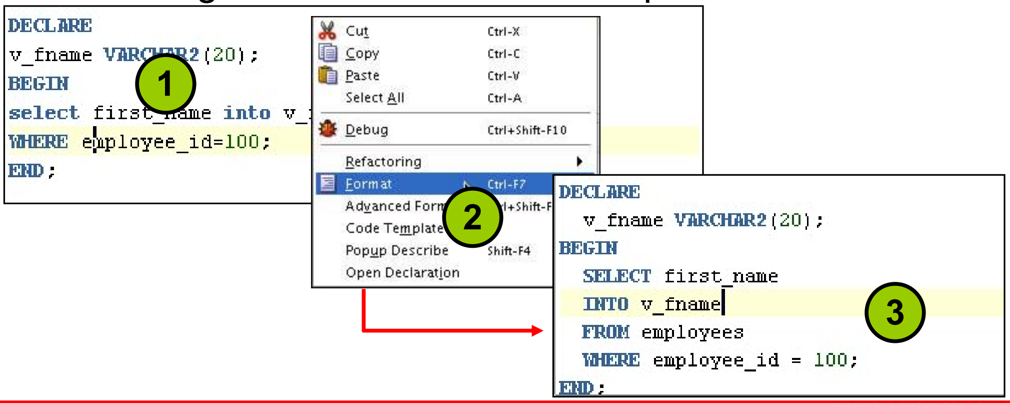
- **Literals:** Any value that is assigned to a variable is a literal. Any character, numeral, Boolean, or date value that is not an identifier is a literal. Literals are classified as:
  - **Character literals:** All string literals have the data type `CHAR` or `VARCHAR2` and are, therefore, called character literals (for example, `John`, and `12c`).
  - **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. However, when you include the explanation in a PL/SQL block, the compiler cannot interpret these instructions. Therefore, 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.
  - Two hyphens (`--`) are used to comment a single line.
  - The beginning and ending comment delimiters (`/*` and `*/`) are used to comment multiple lines.

PL/SQL Block Syntax and Guidelines

- Using Literals
  - Character and date literals must be enclosed in single quotation marks.
  - Numbers can be simple values or in scientific notation.

```
v_name := 'Henderson';
```

- Formatting Code: Statements can span several lines.



```
DECLARE
v_fname VARCHAR2(20);
BEGIN
select first_name into v_
WHERE employee_id=100;
END;
```

```
DECLARE
  v_fname VARCHAR2(20);
BEGIN
  SELECT first_name
  INTO v_fname
  FROM employees
  WHERE employee_id = 100;
END;
```

Using Literals

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

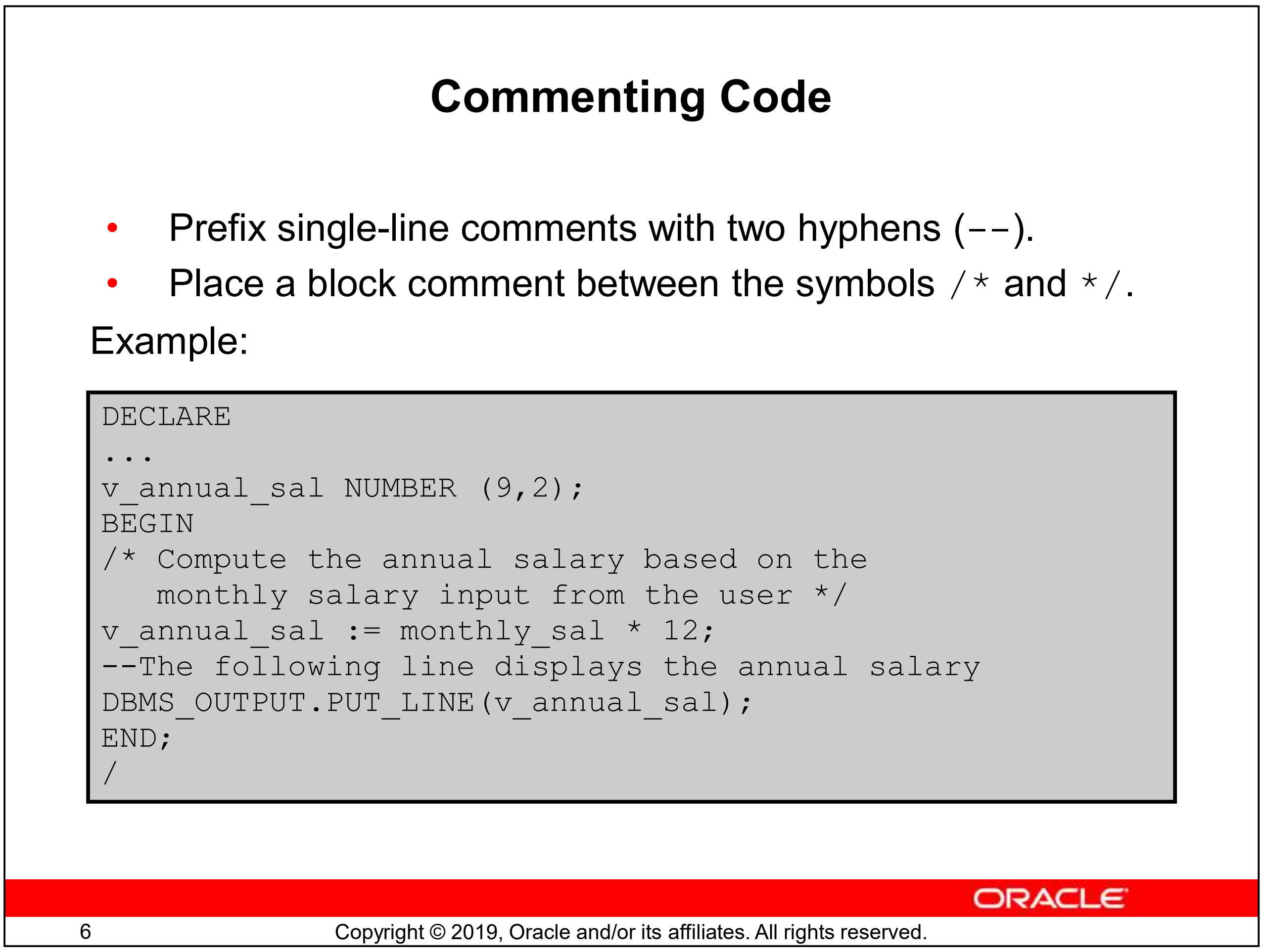
- Character literals include all printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols.
- Numeric literals can be represented either by a simple value (for example, `-32.5`) or in scientific notation (for example, `2E5` means 2 * 10^5 = 200,000).

Formatting Code

In a PL/SQL block, a SQL statement can span several lines (as shown in example 3 in the slide).

You can format an unformatted SQL statement (as shown in example 1 in the slide) by using the SQL Worksheet shortcut menu. Right-click the active SQL Worksheet and, in the shortcut menu that appears, select the Format option (as shown in example 2).

Note: You can also use the shortcut key combination of Ctrl + F7 to format your code.



Commenting Code

- Prefix single-line comments with two hyphens (`--`).
- Place a block comment between the symbols `/*` and `*/`.

Example:

```
DECLARE
...
v_annual_sal NUMBER (9,2);
BEGIN
/* Compute the annual salary based on the
   monthly salary input from the user */
v_annual_sal := monthly_sal * 12;
--The following line displays the annual salary
DBMS_OUTPUT.PUT_LINE(v_annual_sal);
END;
/
```

ORACLE

You should comment code to document each phase and to assist debugging. In PL/SQL code:

- A single-line comment is commonly prefixed with two hyphens (`--`)
- You can also enclose a comment between the symbols `/*` and `*/`

Note: For multiline comments, you can either precede each comment line with two hyphens, or use the block comment format.

Comments are strictly informational and do not enforce any conditions or behavior on the logic or data. Well-placed comments are extremely valuable for code readability and future code maintenance.

SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row functions
- Not available in procedural statements:
  - `DECODE`
  - Group functions

SQL provides several predefined functions that can be used in SQL statements. Most of these functions (such as single-row number and character functions, data type conversion functions, and date and time-stamp 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 are, therefore, 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:

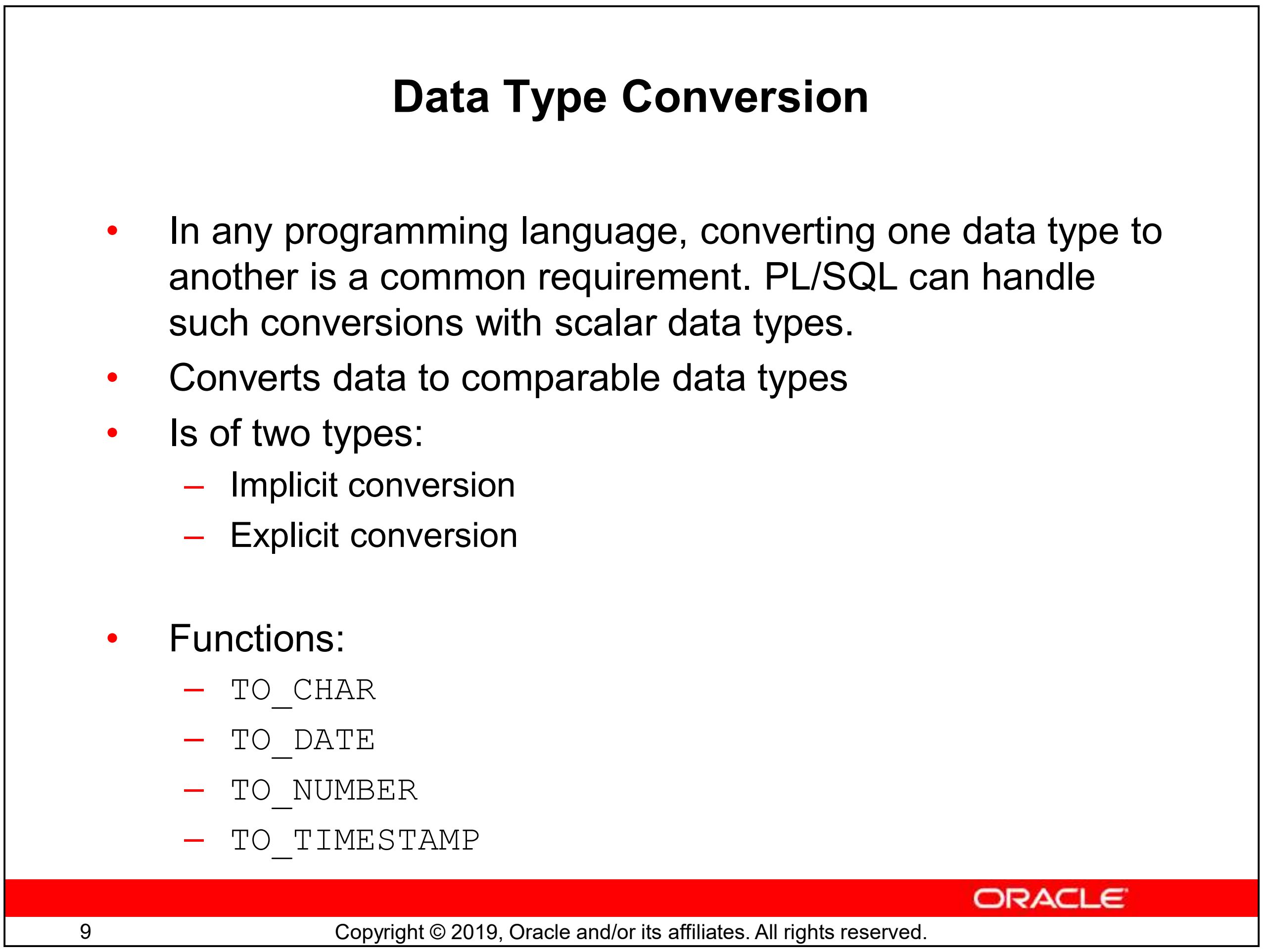
```
v_desc_size INTEGER(5);
v_prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

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

- Get the number of months an employee has worked:

```
v_tenure:= MONTHS_BETWEEN (CURRENT_DATE, v_hiredate);
```

You can use SQL functions to manipulate data. These functions are grouped into the following categories:

- Number
- Character
- Conversion
- Date
- Miscellaneous



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.
- Converts data to comparable data types
- Is of two types:
  - Implicit conversion
  - Explicit conversion
- Functions:
  - `TO_CHAR`
  - `TO_DATE`
  - `TO_NUMBER`
  - `TO_TIMESTAMP`

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:

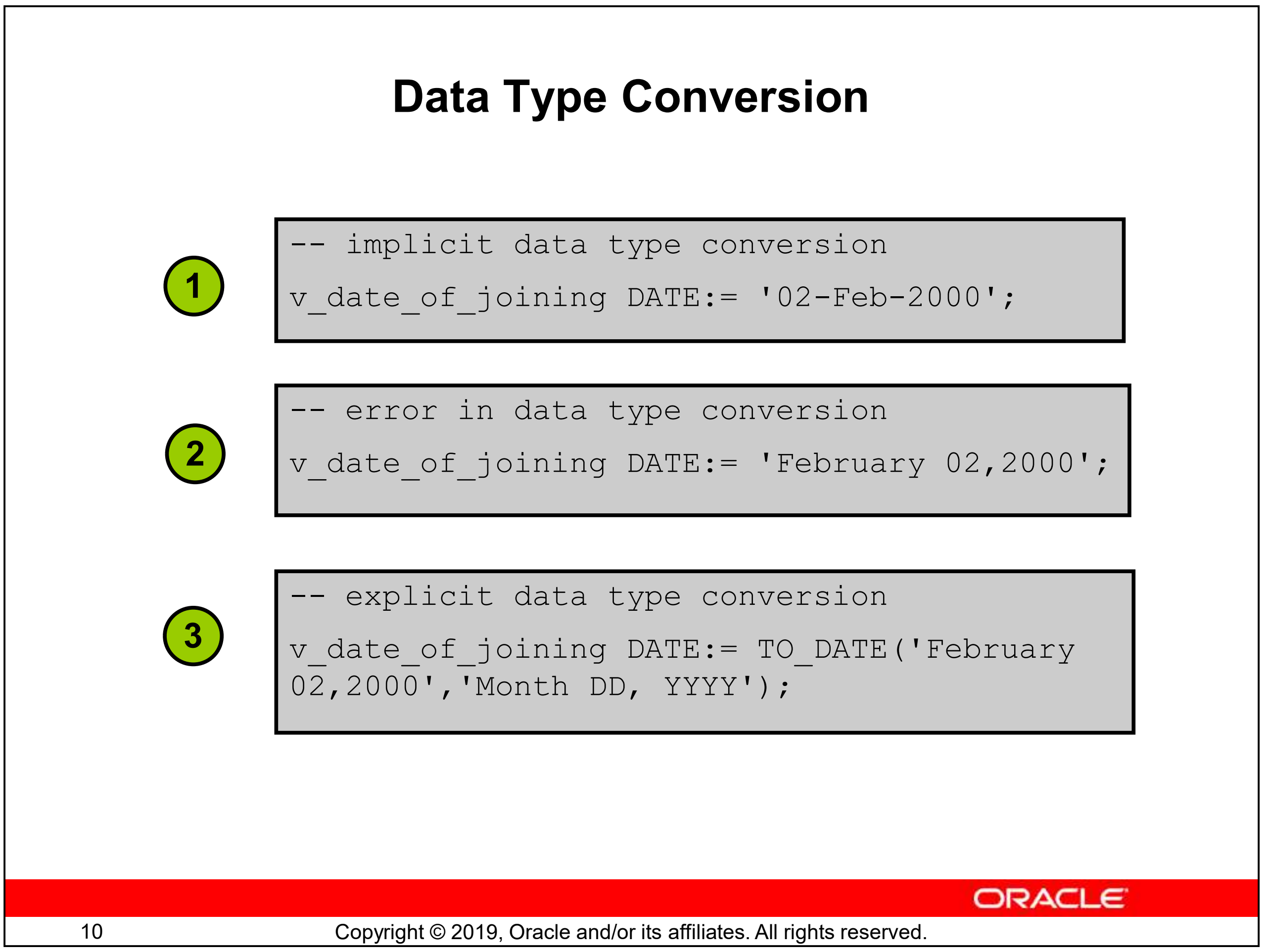
```
DECLARE
 v_salary NUMBER(6):=6000;
 v_sal_hike VARCHAR2(5):='1000';
 v_total_salary v_salary%TYPE;
BEGIN
 v_total_salary:=v_salary + v_sal_hike;
 DBMS_OUTPUT.PUT_LINE(v_total_salary);
END;
/
```

In this example, the `sal_hike` variable is of the `VARCHAR2` type. When 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:

- Characters and numbers
- Characters and dates

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

```
-- implicit data type conversion
v_date_of_joining DATE:= '02-Feb-2000';
```

2

```
-- error in data type conversion
v_date_of_joining DATE:= 'February 02,2000';
```

3

```
-- explicit data type conversion
v_date_of_joining DATE:= TO_DATE('February
02,2000','Month DD, YYYY');
```

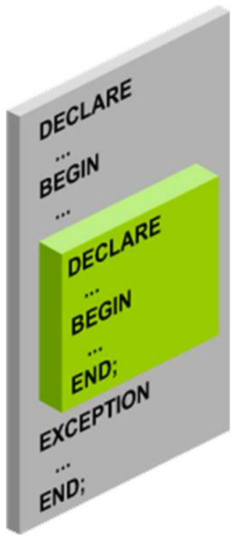
Note the three examples of implicit and explicit conversions of the `DATE` data type in the slide:

1. Because the string literal being assigned to `date_of_joining` is in the default format, this example performs implicit conversion and assigns the specified date to `date_of_joining`.
2. The PL/SQL returns an error because the date that is being assigned is not in the default format.
3. The `TO_DATE` function is used to explicitly convert the given date in a particular format and assign it to the `DATE` data type variable `date_of_joining`.

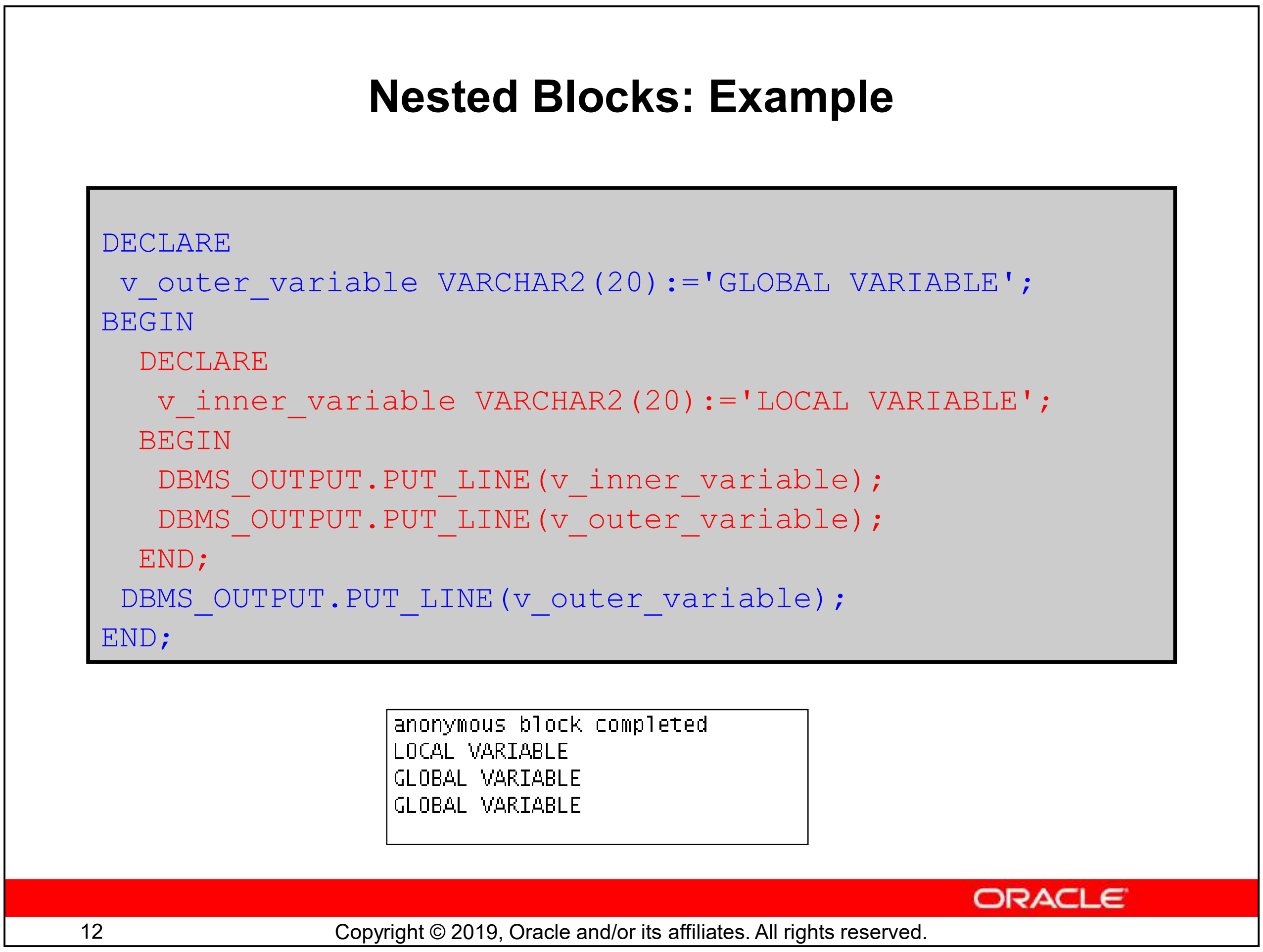
Nested Blocks

PL/SQL blocks can be nested.

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



Being procedural gives PL/SQL 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

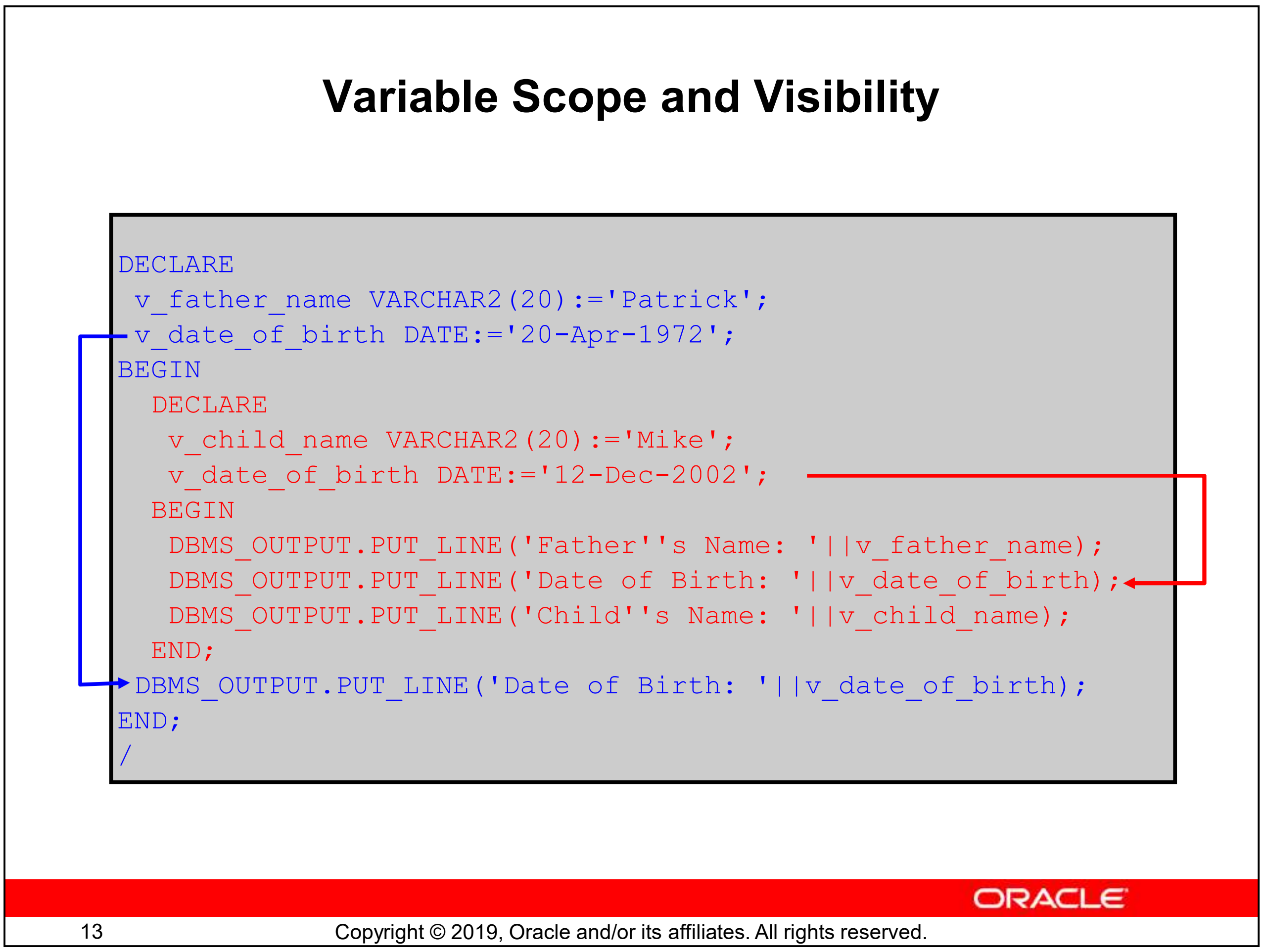
```
DECLARE
 v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
   v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
   DBMS_OUTPUT.PUT_LINE(v_inner_variable);
   DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
 DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

```
anonymous block completed
LOCAL VARIABLE
GLOBAL VARIABLE
GLOBAL VARIABLE
```

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

`v_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, `v_outer_variable` is considered to be 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.

`v_inner_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

```
DECLARE
 v_father_name VARCHAR2(20):='Patrick';
 v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
   v_child_name VARCHAR2(20):='Mike';
   v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
   DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
   DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
  END;
 DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
END;
/
```

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

```
anonymous block completed
Father's Name: Patrick
Date of Birth: 12-DEC-02
Child's Name: Mike
Date of Birth: 20-APR-72
```

Examine the date of birth that is printed for father and child. The output does not provide the correct information, because the scope and visibility of the variables are not applied correctly.

- 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 `v_father_name` variable and the first occurrence of the `v_date_of_birth` variable are declared in the outer block. These variables have the scope of the block in which they are declared. Therefore, the scope of these variables is limited to the outer block.

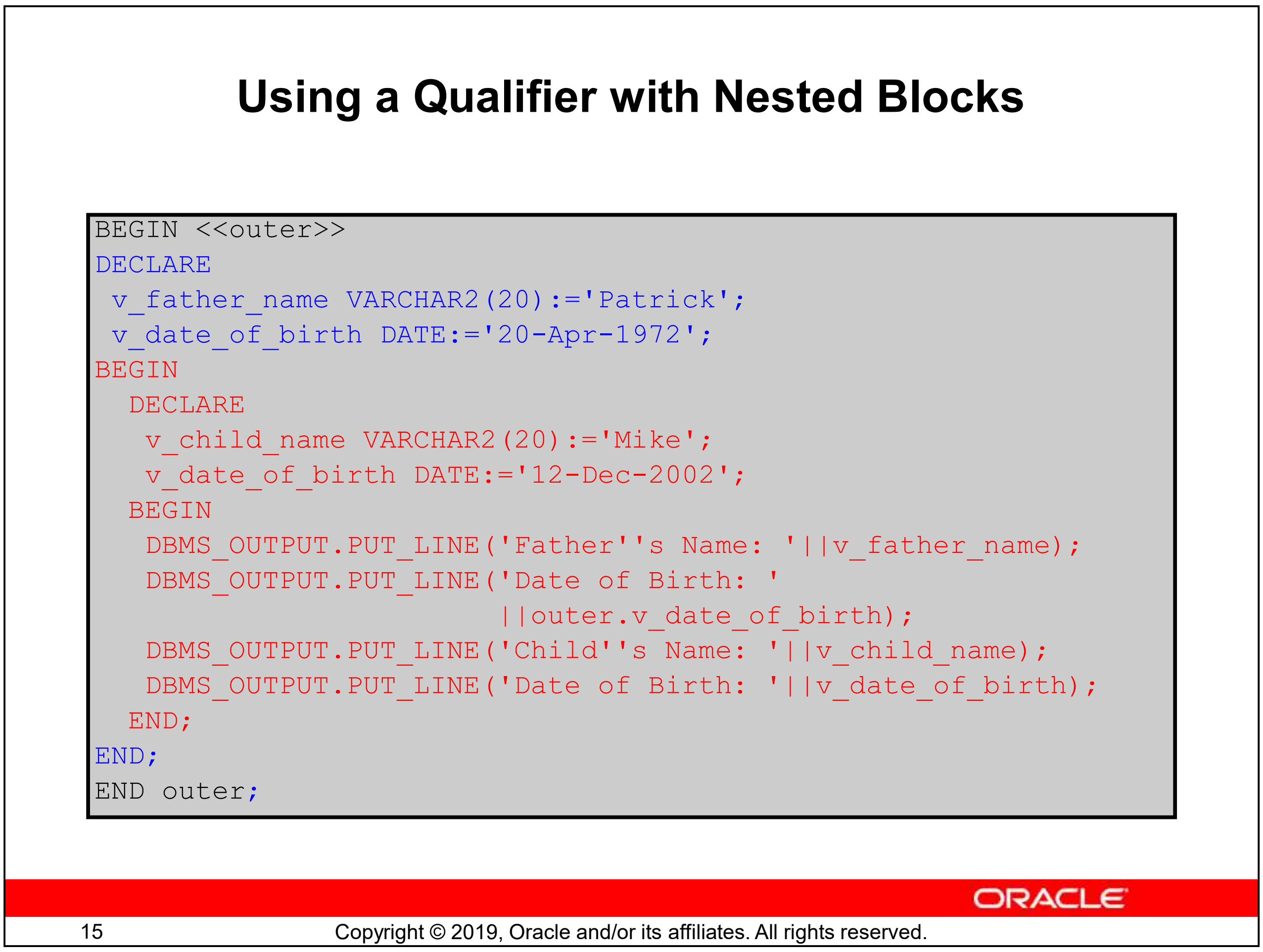
Scope

- The `v_child_name` and `v_date_of_birth` variables 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 `v_date_of_birth` variable declared in the outer block has 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 in the outer block and, therefore, can be printed.

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



Using a Qualifier with Nested Blocks

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

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.

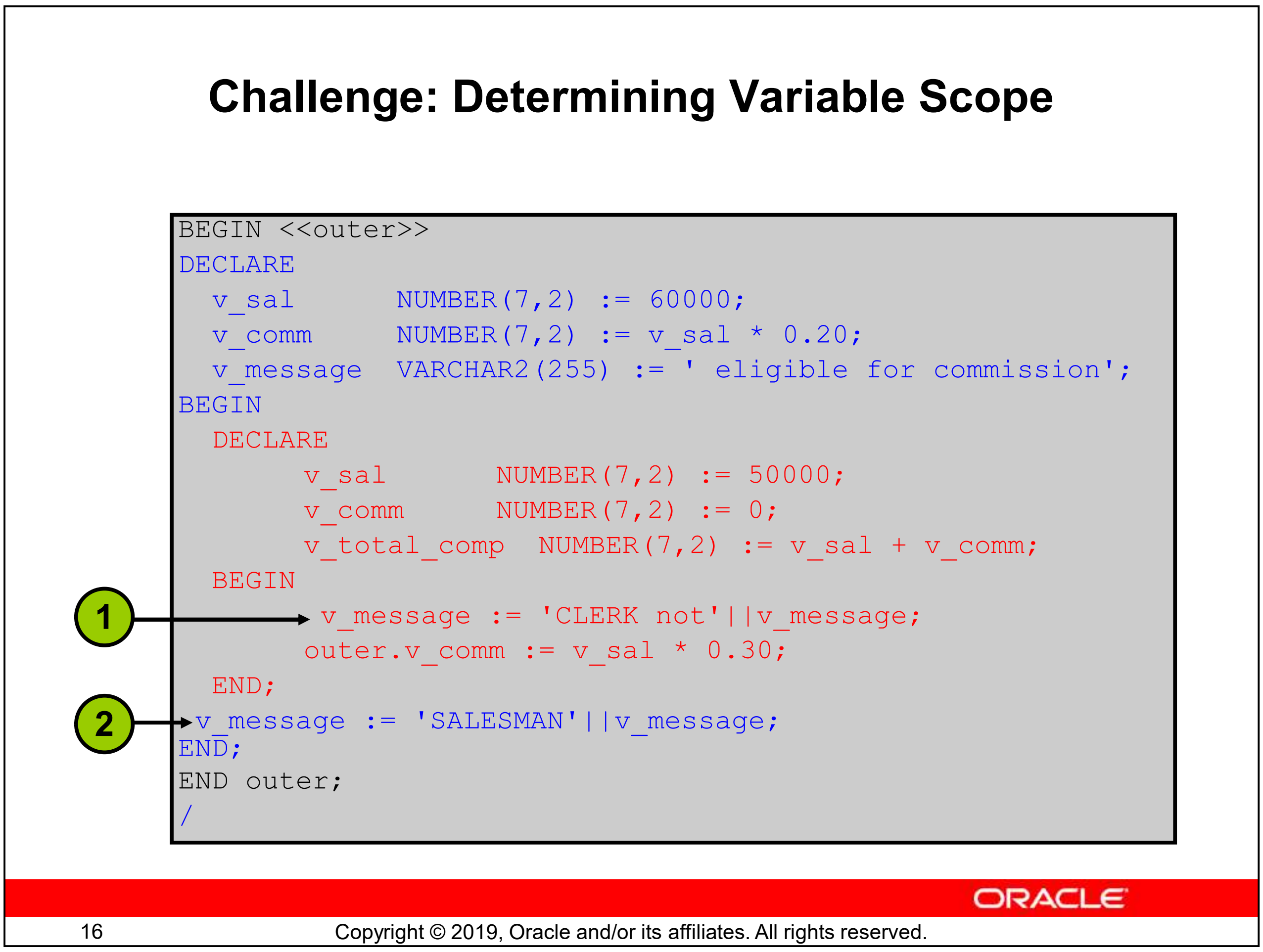
Example

In the code example:

- The outer block is labeled `outer`
- Within the inner block, the `outer` qualifier is used to access the `v_date_of_birth` variable that is declared in the outer block. Therefore, the father's date of birth and the child's date of birth can both be printed from within the inner block.
- The output of the code in the slide shows the correct information:

```
anonymous block completed
Father's Name: Patrick
Date of Birth: 20-APR-72
Child's Name: Mike
Date of Birth: 12-DEC-02
```

Note: Labeling is not limited to the outer block. You can label any block.



Challenge: Determining Variable Scope

```
BEGIN <<outer>>
DECLARE
  v_sal       NUMBER(7,2) := 60000;
  v_comm      NUMBER(7,2) := v_sal * 0.20;
  v_message   VARCHAR2(255) := ' eligible for commission';
BEGIN
  DECLARE
        v_sal        NUMBER(7,2) := 50000;
        v_comm       NUMBER(7,2) := 0;
        v_total_comp   NUMBER(7,2) := v_sal + v_comm;
  BEGIN
         v_message := 'CLERK not'||v_message;
        outer.v_comm := v_sal * 0.30;
  END;
 v_message := 'SALESMAN'||v_message;
END;
END outer;
/
```

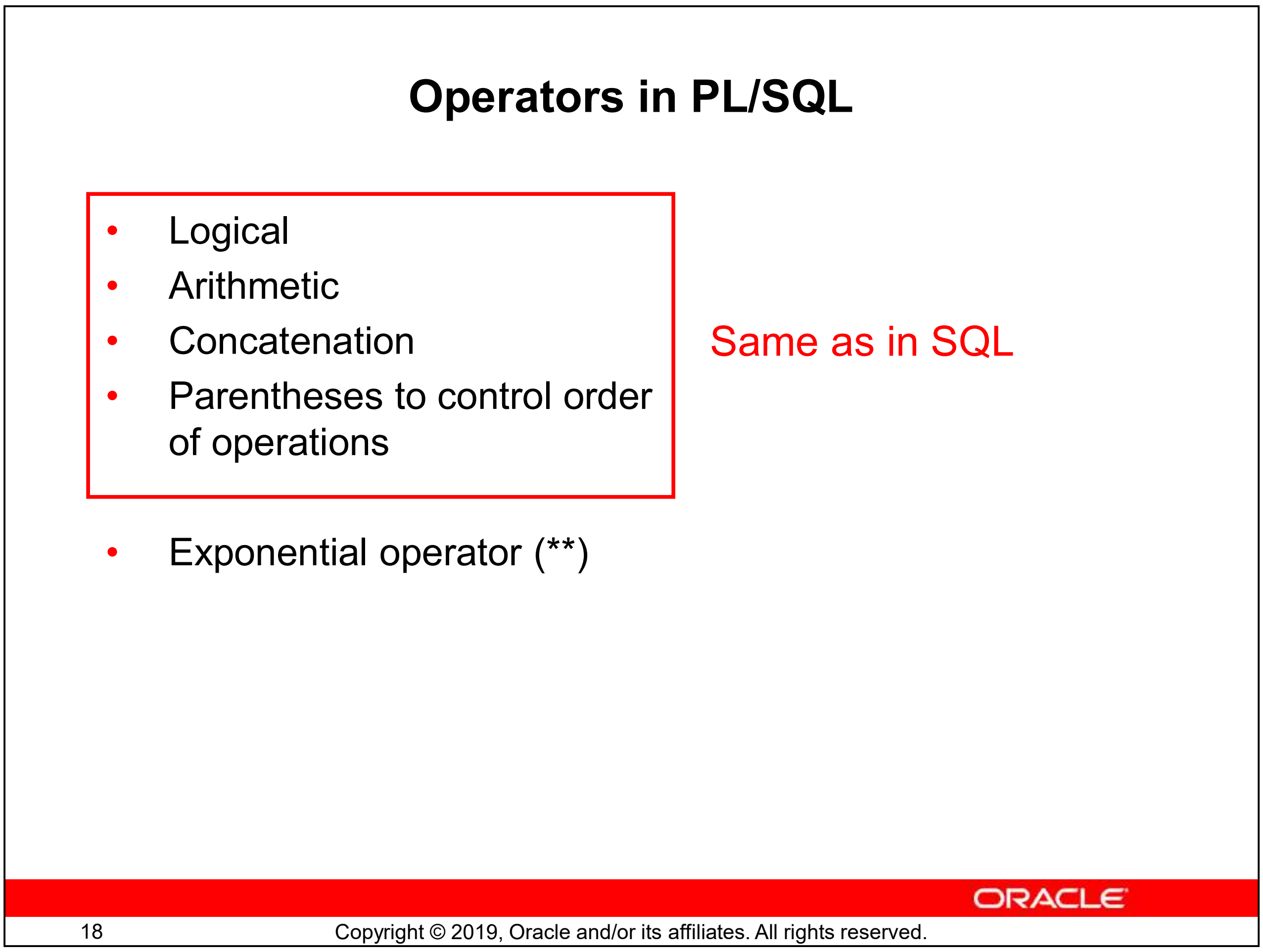
Evaluate the PL/SQL block in the slide. Determine each of the following values according to the rules of scoping:

1. Value of `v_message` at position 1
2. Value of `v_total_comp` at position 2
3. Value of `v_comm` at position 1
4. Value of `outer.v_comm` at position 1
5. Value of `v_comm` at position 2
6. Value of `v_message` at position 2

Answers: Determining Variable Scope

Answers to the questions of scope are as follows:

1. Value of `v_message` at position 1: **`CLERK` not eligible for commission**
2. Value of `v_total_comp` at position 2: **Error. `v_total_comp` is not visible here because it is defined within the inner block.**
3. Value of `v_comm` at position 1: **0**
4. Value of `outer.v_comm` at position 1: **12000**
5. Value of `v_comm` at position 2: **15000**
6. Value of `v_message` at position 2: **`SALESMANCLERK` not eligible for commission**



Operators in PL/SQL

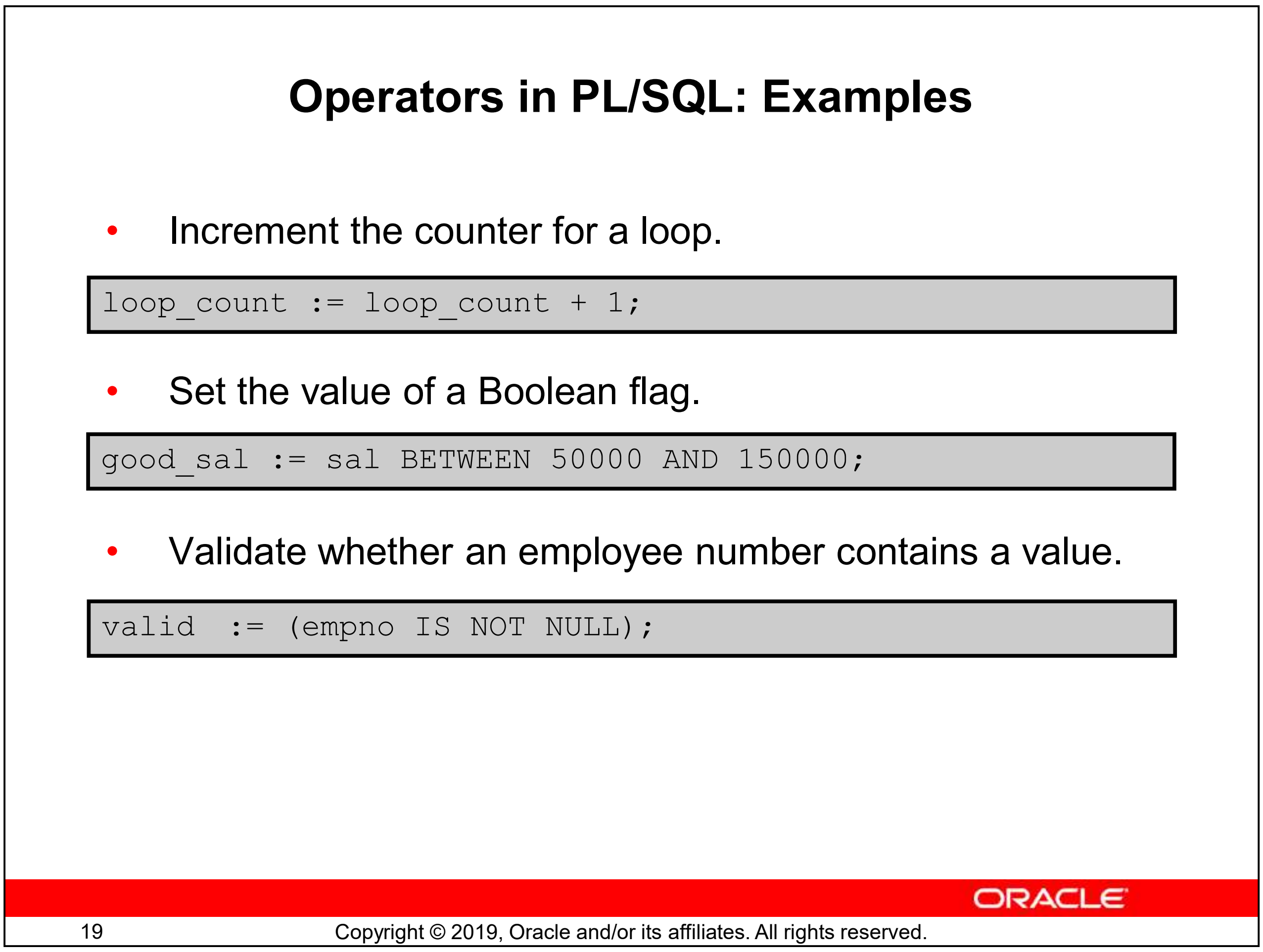
- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations

Same as in SQL

- Exponential operator (**)

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.

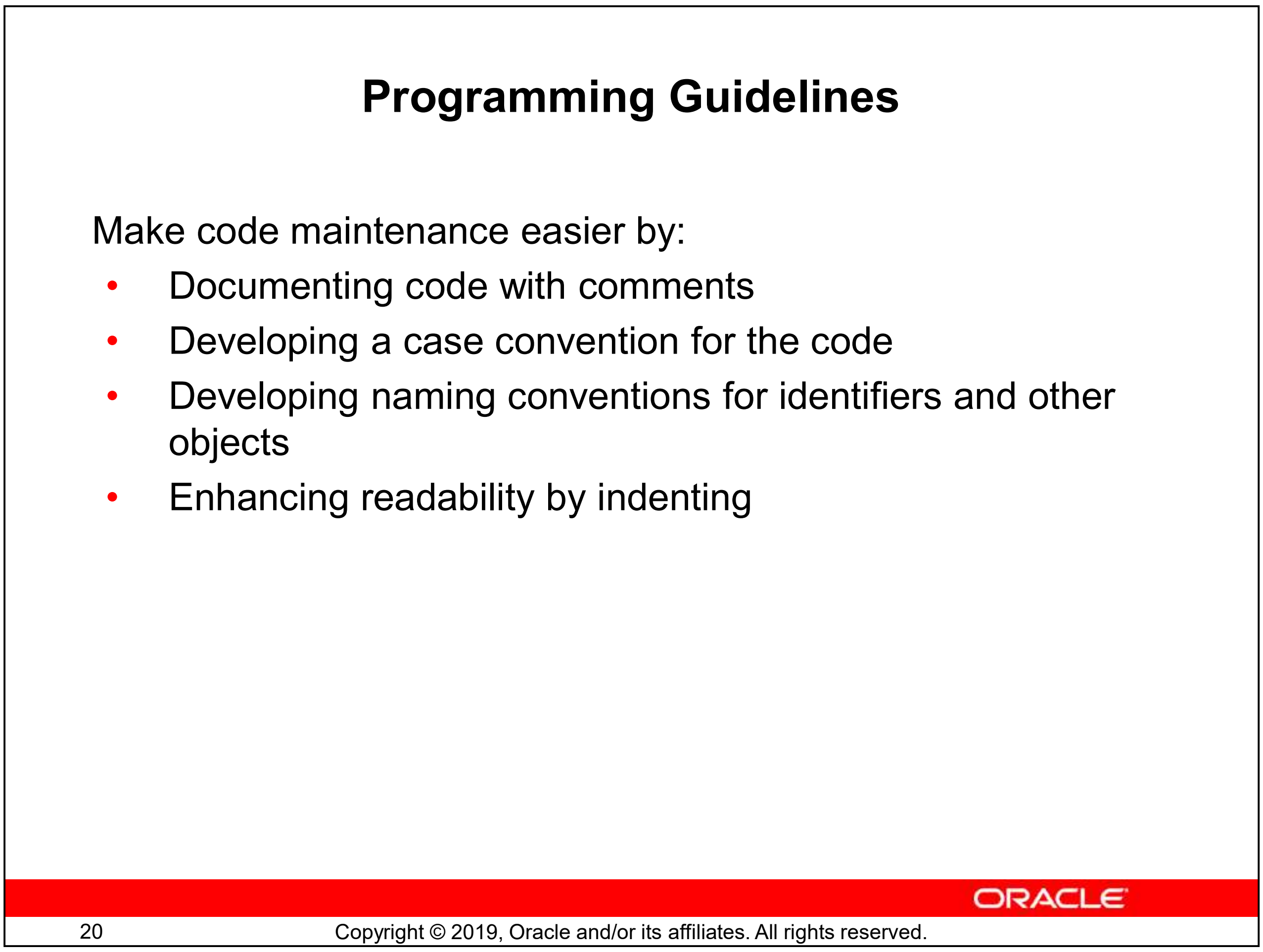
```
good_sal := sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number contains a value.

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

When you are 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

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	Lowercase, plural	employees, departments
Database columns	Lowercase, singular	employee_id, department_id

Indenting Code

For clarity, indent each level of code.

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

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

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;
```

Quiz

You can use most SQL single-row functions such as number, character, conversion, and date single-row functions in PL/SQL expressions.

a. True

b. False

Answer: a

SQL Functions in PL/SQL

SQL provides several predefined functions that can be used in SQL statements. Most of these functions (such as single-row number and character functions, data type conversion functions, and date and time-stamp 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 are, therefore, available only in SQL statements in a PL/SQL block. The functions mentioned here are only a subset of the complete list.

Summary

In this lesson, you should have learned how to:

- Identify lexical units in a PL/SQL block
- 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
- Use sequences in PL/SQL expressions

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 a part of the executable section, it can also contain nested blocks. 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 with 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.