

Oracle Academy

Introduction to Database Programming with PL/SQL

Instructor Resource Guide

INSTRUCTOR NOTES FOR SLIDES

SECTION 6 LESSON 1 – Handling Exceptions

Slide 1: Handling Exceptions

No instructor notes for this slide

Slide 2: What Will I Learn?

No instructor notes for this slide

Slide 3: Why Learn It?

You are going to learn how to handle errors so that an abrupt end of program does not occur. Programs should be managed so that even unanticipated errors are handled, no program should ever just ‘bomb’ out with an error.

Slide 4: Tell Me / Show Me – What is an Exception?

Ask the students to discuss errors that they may have seen in an application. The error can be a handled or unhandled error. Below are some examples:

- Entering an incorrect username and/or password
- Forgetting to include the @ in an email address
- Entering a credit card number incorrectly
- Entering an expiration date that has passed
- Selecting more than one row into a single variable
- Receiving “no rows returned” from a select statement

Slide 5: Tell Me / Show Me – Exceptions in PL/SQL

To confirm the spelling of Republic of Korea in the data, run a statement such as the following:

```
SELECT country_name
FROM wf_countries
WHERE country_name like '%Korea%';
```

Slide 6: Tell Me / Show Me – Exceptions in PL/SQL (continued)

No instructor notes for this slide

Slide 7: Tell Me / Show Me – Exceptions in PL/SQL (continued)

Remind students that a SELECT statement in PL/SQL must return exactly one row. This statement returns no rows and therefore raises an exception in the executable section

Slide 8: Tell Me / Show Me – What is an Exception Handler?

Exception handling is code written to handle errors that occurs during execution. PL/SQL programs will start to get longer and more complicated due to the exception handling code needed to handle all errors, but that is preferable to programs just ‘bombing’ out with unhandled exceptions.

Slide 9: Tell Me / Show Me – What is an Exception Handler? (continued)

These are the more frequent types of errors that may occur during the execution of a program. There are many errors and all should be handled for a PL/SQL program to run smoothly and efficiently.

Point out that one error can often cause another. For example, employee_id 100 already exists in the EMPLOYEES table. A user tries to add a new employee and mistakenly types an employee id of 100. The program then tries to INSERT this new employee and violates a primary key constraint. This is an example of a user action errors causing a data error.

Slide 10: Tell Me / Show Me – Why is Exception Handling Important?

No instructor notes for this slide

Slide 11: Tell Me / Show Me – Why is Exception Handling Important? (continued)

Programs are written to support or enhance the existing systems. If unhandled errors occur and the program execution is stopped, users are frequently confused and might eventually be angry and frustrated with the systems they are trying to use. So, errors need to be handled within the code to keep the execution of the program running continuously.

Slide 12: Tell Me / Show Me – Handling Exceptions with PL/SQL

A block always terminates when PL/SQL raises an exception, but you can specify an exception handler to perform final actions before the block ends.

Slide 13: Tell Me / Show Me – Handling Exceptions with PL/SQL (continued)

Explain the terms used here. When an error occurs, we say that an exception has been raised. A handler is a specific WHEN clause in the EXCEPTION section.

Slide 14: Tell Me / Show Me – Handling Exceptions with PL/SQL (continued)

Stress that when an exception is raised, the rest of the executable section of the block is NOT executed; instead, the EXCEPTION section is searched for a suitable handler.

Slide 15: Tell Me / Show Me – Handling Exceptions with PL/SQL (continued)

There is no exception handler in this block, therefore the block terminates unsuccessfully, returning an ‘unhandled exception’ status code to the calling environment (Application Express), which then reports the exception as shown.

Slide 16: Tell Me / Show Me – Handling Exceptions with PL/SQL (continued)

In this case we have successfully handled the exception inside the block, so PL/SQL returns a ‘success’ status code to the calling environment, which therefore will report ‘Statement Processed’.

Slide 17: Tell Me / Show Me – Trapping Exceptions

No instructor notes for this slide

Slide 18: Tell Me / Show Me – Trapping Exceptions (continued)

No instructor notes for this slide

Slide 19: Tell Me / Show Me – Trapping Exceptions (continued)

There are two types of exceptions in PL/SQL. “Predefined exception” and “User-defined exception”. Predefined exceptions are ones that are built into the language, where Oracle has already associated a known Oracle error number with a named exception. Often used predefined exceptions are TOO_MANY_ROWS and NO_DATA_FOUND, and you can most like guess what they do and when they are raised, just from their name. User-defined exceptions are created by the programmer in the DECLARE section, raised by the programmer in the executable section and handled by the programmer in the exception section. Students will learn more about this in the next two lessons.

Slide 20: Tell Me / Show Me – The OTHERS Exception Handler

No instructor notes for this slide

Slide 21: Tell Me / Show Me – The OTHERS Exception Handler (continued)

No instructor notes for this slide

Slide 22: Tell Me / Show Me – Guidelines for Trapping Exceptions

No matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data, so you do need to pay special care and attention to when your program issues commit or rollback statements.

Slide 23: Tell Me / Show Me – Guidelines for Trapping Exceptions (continued)

No instructor notes for this slide

Slide 24: Tell Me / Show Me – Terminology

Exception – Occurs when an error is discovered during the execution of a program that disrupts the normal operation of the program.

Exception Handler – Code that defines the recovery actions to be performed when execution-time errors occur.

Slide 25: Summary

No instructor notes for this slide

Slide 26: Try It / Solve It

No instructor notes for this slide

SECTION 6 LESSON 2 – Trapping Oracle Server Exceptions

Slide 1: Trapping Oracle Server Exceptions

No instructor notes for this slide

Slide 2: What Will I Learn?

No instructor notes for this slide

Slide 3: Why Learn It?

No instructor notes for this slide

Slide 4: Tell Me / Show Me – Exception Types

An Oracle Server error is an error which is recognized and raised automatically by the Oracle server. There are many thousands of such possible errors, and each one has a predefined error number and error message.

User-defined errors (raised explicitly by the PL/SQL programmer) are explained in the next lesson, but if students are curious at this point, here is an example:

We want to INSERT a new row into DEPARTMENTS, but want to check in advance if the new department_id (a unique column) is already used (this is often done in real applications). We could code:

```
DECLARE
    v_dept_id    departments.department_id%TYPE := 10; -- This department exists
                already
    ...
BEGIN
    SELECT department_id INTO v_dept_id
        FROM departments WHERE department_id = v_dept_id;
    ...
    INSERT INTO departments VALUES (v_dept_id, ....);
    ...
```

Because department 10 already exists, the SELECT successfully returns exactly one row and therefore no Oracle Server exception is raised, and the Oracle Server attempts to continue executing the rest of the block. But clearly we do not want to execute the INSERT in this case; instead, we want to display an error message to the user. We would declare and raise our own user_defined (or programmer_defined) exception.

Slide 5: Tell Me / Show Me – Handling exceptions with PL/SQL

No instructor notes for this slide

Slide 6: Tell Me / Show Me – Two Types of Oracle Server Errors

No instructor notes for this slide

Slide 7: Tell Me / Show Me – Trapping Predefined Oracle Server Errors

An Oracle Server error is an error which is recognized and raised automatically by the Oracle server. There are many hundreds of such possible errors, and each one has a predefined error number and error message, for example ORA-01422: exact fetch returns more than requested number of rows.

The most common 20 or so server errors have predefined PL/SQL exception names, for example TOO_MANY_ROWS (= ORA-01422), NO_DATA_FOUND (= ORA-01403).

Slide 8: Tell Me / Show Me – Trapping Predefined Oracle Server Errors (continued)

No instructor notes for this slide

Slide 9: Tell Me / Show Me – Trapping Several Predefined Oracle Server Errors

This example handles TOO_MANY_ROWS and NO_DATA_FOUND errors, with a WHEN OTHERS handler in case any other error occurs.

Remind students that a WHEN OTHERS handler will handle **all** types of raised exceptions: predefined,, non-predefined and user_defined.

Slide 10: Tell Me / Show Me – Trapping Non-Predefined Oracle Server Errors

No instructor notes for this slide

Slide 11: Tell Me / Show Me – Trapping Non-Predefined Oracle Server Errors (continued)

No instructor notes for this slide

Slide 12: Tell Me / Show Me – Non-Predefined Error

Point out that the EXCEPTION section can refer to exceptions only by name, not by number (i.e. we cannot code WHEN ORA-01400 THEN). There is no predefined PL/SQL name for the ORA-1400 exception, so we need to create our own name for it. The next slide shows how to do this.

Slide 13: Tell Me / Show Me – Non-Predefined Error (continued)

No instructor notes for this slide

Slide 14: Tell Me / Show Me – Non-Predefined Error (continued)

No instructor notes for this slide

Slide 15: Tell Me / Show Me – Non-Predefined Error (continued)

No instructor notes for this slide

Slide 16: Tell Me / Show Me – Functions for Trapping Exceptions

Students may ask: why is the SQLCODE for NO_DATA_FOUND +100 instead of -01403?

Answer: some of these error codes are used in other (non-Oracle) databases, and +100 is an internationally-agreed code when no rows are returned from a query.

Slide 17: Tell Me / Show Me – Functions for Trapping Exceptions (continued)

SQLCODE and SQLERRM are often used in a WHEN OTHERS handler. Someone (often the Database Administrator) would be responsible for reading the ERROR_LOG table and taking suitable action.

Ask students: why can't we use SQLCODE or SQLERRM directly in a SQL statement?

Answer: because that SQL statement (the INSERT INTO error_log in the slide) could also raise an exception, which would change the values of SQLCODE and SQLERRM.

It is good programming practice to capture SQLERRM and SQLCODE in a WHEN OTHERS and insert the returned values into an error table, so you can later go back and see what kind of problems your program encountered, and if the same errors shows up a number of times, then include a specific handler for it.

Review:

What happens when the user enters an invalid department? You need to define a predefined-user error to raise an error.

1. Declaring the name of the user-defined exception within the declarative section.
2. Using the RAISE statement to raise the exception explicitly within the executable section.
3. Referencing the declared exception within the corresponding exception-handling routine.

The RAISE statement can be used to raise a user defined exception.

Slide 18: Tell Me / Show Me – Terminology

Predefined Oracle Server Errors: Each of these has a predefined name. For example, if the error ORA-01403 occurs when no rows are retrieved from the database in a SELECT statement, then PL/SQL raises the predefined exception-name NO_DATA_FOUND.

Non-predefined Oracle Server Errors: Each of these has a standard Oracle error number (ORA-nnnnn) and error message, but not a predefined name. We declare our own names for these so that we can reference these names in the exception section.

PRAGMA EXCEPTION_INIT – Tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any Oracle Server exception by name and to write a specific handler for it.

SQLERRM – Returns character data containing the message associated with the error number

SQLCODE – Returns the numeric value for the error code (You can assign it to a NUMBER variable.)

Slide 19: Summary

No instructor notes for this slide

Slide 20: Try It / Solve It

No instructor notes for this slide

SECTION 6 LESSON 3 – Trapping User-Defined Exceptions

Slide 1: Trapping User-Defined Exceptions

No instructor notes for this slide

Slide 2: What Will I Learn?

No instructor notes for this slide

Slide 3: Why Learn It?

Ask students to think of situations where the Oracle server would execute a statement successfully (and therefore not raise an exception automatically) but there is still an “error” from the user’s viewpoint. Possible examples:

- A DML UPDATE or DELETE statement modifies no rows
- A SELECT statement successfully reads a row which should not exist yet
- A Stock Clerk has been identified as a manager, but our business rules state that clerks cannot be managers.

Slide 4: Tell Me / Show Me – Exception types

No instructor notes for this slide

Slide 5: Tell Me / Show Me – Trapping User-Defined Exceptions

No instructor notes for this slide

Slide 6: Tell Me / Show Me – Trapping User-Defined Exceptions (continued)

Remind students that an UPDATE or DELETE DML statement is treated as successful by the server even if it modifies no rows. Therefore the Oracle server will not automatically raise an exception in this case. If we want to raise an exception, we must do it ourselves.

Slide 7: Tell Me / Show Me – Trapping User-Defined Exceptions (continued)

No instructor notes for this slide

Slide 8: Tell Me / Show Me – Trapping User-Defined Exceptions (continued)

No instructor notes for this slide

Slide 9: Tell Me / Show Me – Trapping User-Defined Exceptions (continued)

No instructor notes for this slide

Slide 10: Tell Me / Show Me – Trapping User-Defined Exceptions (continued)

Remind students that when any kind of exception is raised, the rest of the executable section is not executed. Therefore in the slide example, if the UPDATE modifies no rows, the COMMIT will not be executed.

Instead of using IF SQL%NOTFOUND THEN ... we could have coded: IF SQL%ROWCOUNT = 0 THEN ...

Slide 11: Tell Me / Show Me – Trapping User-Defined Exceptions (continued)

No instructor notes for this slide

Slide 12: Tell Me / Show Me – The RAISE Statement

No instructor notes for this slide

Slide 13: Tell Me / Show Me – The RAISE_APPLICATION_ERROR Procedure

No instructor notes for this slide

Slide 14: Tell Me / Show Me – The RAISE_APPLICATION_ERROR Procedure (continued)

No instructor notes for this slide

Slide 15: Tell Me / Show Me – The RAISE_APPLICATION_ERROR Procedure (continued)

No instructor notes for this slide

Slide 16: Tell Me / Show Me – RAISE_APPLICATION_ERROR in the Executable Section

Note that an error raised by RAISE_APPLICATION_ERROR is an unhandled exception which is propagated back to the calling environment. The whole idea is to allow the calling application to display a business-meaningful error message to the user. If the exception was handled successfully within the PL/SQL block, the application would not see the error at all.

Slide 17: Tell Me / Show Me – RAISE_APPLICATION_ERROR in the Exception Section

Ask students to imagine that another member of their family is using a computer and sees an error message on the screen. Which of these messages would their family member rather see?

ORA-01403: no data found

ORA-20201: This manager has no employees

Slide 18: Tell Me / Show Me – Using RAISE_APPLICATION_ERROR with a User-Defined Exception

In this example, when an invalid last name is entered, the user-defined exception **e_name** is raised using its error number -20999. Because the EXCEPTION_INIT pragma was used to define the error name e_name and its error code -20999, when the error code is raised, the exception handler for e_name is run and the exception is not propagated to the calling environment

Slide 19: Tell Me / Show Me – Terminology

User-Defined errors - These errors are not automatically raised by the Oracle Server, but are defined by the programmer and are specific to the programmer's code.

RAISE – Use this statement to raise a named exception.

RAISE_APPLICATION_ERROR – A procedure used to return user-defined error messages from stored subprograms.

Slide 20: Summary

No instructor notes for this slide

Slide 21: Try It / Solve It

No instructor notes for this slide

SECTION 6 LESSON 4 – Recognizing the Scope of Variables

Slide 1: Recognizing the Scope of Variables

This lesson is partly a review of section 2 Lesson 8 (which introduced nested blocks, scope of variables and exception propagation) plus some new material. This review is important because since Section 2, students have learned more about exception handling.

Slide 2: What Will I Learn?

No instructor notes for this slide

Slide 3: Why Learn It?

No instructor notes for this slide

Slide 4: Tell Me / Show Me – Review of Nested Blocks

A PL/SQL block nested within another PL/SQL block may be called a nested block, and enclosed block, a child block or a sub-block.

A PL/SQL block that calls another PL/SQL block, anonymous or named, may be referred to as either the enclosing block or the parent block.

Slide 5: Tell Me / Show Me – Review of Nested Blocks

No instructor notes for this slide

Slide 6: Tell Me / Show Me – Review of Variable Scope

The general advantage of a nested block is that you create a scope for all the declared objects and executable statements in that block. You can use this scope to improve your control over activity in your program. This also means that if an error occurs in a block, then it is just in this block execution is halted, once the error has been handled locally, program control continues outside that block.

CURSORs are a type of variable, and subject to the same scoping rules as more “obvious” variables.

Answer: The scope of `v_outer_variable` includes both the outer and inner blocks. The scope of `v_inner_variable` includes only the inner block. It is valid to refer to `v_outer_variable` within the inner block, but referencing `v_inner_variable` within the outer block would return an error.

Slide 7: Tell Me / Show Me – Review of Variable Scope (continued)

Answer: The scope of `v_father_name` and `v_date_of_birth` includes both the outer and inner blocks. The scope of `v_child_name` includes only the inner block. Referencing `v_child_name` in the outer block would return an error.

Slide 8: Tell Me / Show Me – Review of Variable Scope (continued)

Remind students that a cursor is a type of variable, and is subject to the same scoping rules as more “obvious” variables. `emp_curs` was declared in the inner block, and therefore cannot be referenced by the `CLOSE` statement in the outer block

Slide 9: Tell Me / Show Me – Review of Variable Scope (continued)

This time the code will work, because the cursor is declared in the outer block, and can therefore be referenced both in the outer block (OPEN, CLOSE) and in the inner block (FETCH, %NOTFOUND).

Slide 10: Tell Me / Show Me – How does PL/SQL Resolve the Names of Variables?

Therefore the scope of a variable consists of all the blocks in which the variable is either local (the declaring block) or global (nested blocks within the declaring block).

Slide 11: Tell Me / Show Me – Three Levels of Nested Block

The scope of v_outervar is all three blocks, while the scope of v_middlevar is the middle and innermost blocks. We can reference v_middlevar in the middle or innermost blocks, but not in the outer block.

Slide 12: Tell Me / Show Me – Review of Variable Naming

Answer: YES because although the two variables have the same name, they are declared in different blocks. They could even have different datatypes.

Slide 13: Tell Me / Show Me – Review of Variable Naming (continued)

The statement will reference the v_date_of_birth declared in the inner block.

Slide 14: Tell Me / Show Me – Review of Variable Visibility

Two of the most important concepts related to a PL/SQL block are those of the scope and visibility of identifiers. An identifier is the name of a PL/SQL object, which could be anything from a variable to a program name. In order to manipulate a PL/SQL identifier you have to be able to reference that identifier in such a way that the code will compile.

The scope of an identifier is the part of a program in which you can make a reference to the identifier and have that reference resolved by the compiler. An identifier is visible in a program when it can be referenced using an unqualified name.

A qualifier for an identifier can be a package name, procedure or function, or loop label. You qualify the name of an identifier with dot notation, the same way you would qualify a column name with the name of its table.

1. Observe 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 here.
2. The father's date of birth is visible here and therefore can be printed.

Slide 15: Tell Me / Show Me – Review of Variable Visibility (continued)

No instructor notes for this slide

Slide 16: Tell Me / Show Me – Review of Block Labels

Anonymous blocks don't have names under which they can be stored in the database. By using block labels, you can give a name to your block for the duration of its execution. A block label is a PL/SQL label which is placed directly in front of the first line of the block. You might use block labels for either of these reasons:

1. Improve the readability of your code
2. Qualify the names of elements declared in the block to distinguish them from similar named elements from other blocks.

Slide 17: Tell Me / Show Me – Using Block Labels to Gain Variable Visibility

We could also label the inner block but this is not needed here. In fact we can label a non-nested (single) block, although there is no need to do it.

Slide 18: Tell Me / Show Me – Exception Handling in Nested Blocks

No instructor notes for this slide

Slide 19: Tell Me / Show Me – Propagating Exceptions to an Outer Block

No instructor notes for this slide

Slide 20: Tell Me / Show Me – Propagating Exceptions to an Outer Block (continued)

Point out that an EXCEPTION variable follows the same scoping rules as other variables. In this example, the user-defined exception e_no_rows is declared in the outer block but raised in the inner block.

Slide 21: Tell Me / Show Me – Propagating Exceptions from a Subblock

You cannot declare an exception twice in the same block. You can declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.

Slide 22: Tell Me / Show Me – Propagating Predefined Oracle Server Exceptions from a Subblock

Answer: the SELECT raises a NO_DATA_FOUND exception in the inner block. This is propagated to the outer block which handles it successfully using WHEN OTHERS. Only Message 4 will be displayed

Slide 23: Tell Me / Show Me – Propagating User-named Exceptions from a Subblock

Answer: a compile-time error will occur and the block will not be executed at all, so nothing is displayed. Why? Because e_myexcep is a variable. It is declared within the inner block and is out of scope and cannot be referenced in the outer block.

Slide 24: Tell Me / Show Me – Scope of Exception Names

When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, the exception propagates. That is, the exception will continue to look for a block

which handles that exception in successive enclosing blocks until a handler is found or there are no more blocks to search.

Slide 25: Tell Me / Show Me – Terminology

Scope – The portion of a program in which the variable is declared and is accessible.

Visibility – The portion of the program where the variable can be accessed without using a qualifier.

Qualifier – A label given to a block.

Slide 26: Summary

No instructor notes for this slide

Slide 27: Try It / Solve It

No instructor notes for this slide

PRACTICE SOLUTIONS

SECTION 6 LESSON 1 – Handling Exceptions

Terminology

1. Exception Handler Code that defines the recovery actions to be performed when execution-time errors occur.
2. Exception Occurs when an error occurs during the execution of a program that disrupts the normal operation of the program.

Try It / Solve It

1. What happens when Oracle encounters a runtime problem while executing a PL/SQL block?

An exception is raised and the rest of the block's execution section is not executed.

2. What do you need to add to your PL/SQL block to address these problems?

Add an exception section which includes handlers to trap and deal with all possible problems.

3. List three advantages of handling exceptions within a PL/SQL block.

It can protect the user from frequent repeated errors.

It can prevent data in the database from being lost or overwritten.

PL/SQL code is more readable because error-handling routines can be written in the same block in which the error occurred.

4. Run this PL/SQL code and then answer the questions that follow.

```
DECLARE
  v_jobid employees.job_id%TYPE;
BEGIN
  SELECT job_id
  INTO v_jobid
  FROM employees
  WHERE department_id = 80;
END;
```

- A. What happens when you run the block?

The block fails because there is more than one employee in department 80 and an exception (error) is raised.

B. In your own words, explain what can you do to fix this problem.

Select a unique column for the WHERE clause (the block could still fail if the unique value entered did not exist;

Write an exception handler to trap the error.

C. Modify the code to fix the problem. Use a TOO_MANY_ROWS exception handler.

```
DECLARE
  v_jobid employees.job_id%TYPE;
BEGIN
  SELECT job_id
    INTO v_jobid
   FROM employees
  WHERE department_id = 80;
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE ('Too many rows returned from the query');
END;
```

D. Run your modified code. What happens this time?

The exception is successfully trapped and the message “Too many rows returned from the query” is displayed.

5. Run the following PL/SQL block, which tries to insert a new row (with department_id = 50) into the departments table. What happens and why?

```
BEGIN
  INSERT INTO departments (department_id, department_name,
                           manager_id, location_id)
    VALUES (50, 'A new department', 100, 1500);
  DBMS_OUTPUT.PUT_LINE('The new department was inserted');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('An exception has occurred.');
```

END;

The INSERT statement failed because department_id 50 already exists and department_id is a unique column. The rest of the executable section was skipped, therefore the message ‘The new department was inserted’ was not displayed. The WHEN OTHERS exception handler was executed.

6. Enter the following PL/SQL block, which tries to SELECT all the employees in a specific department. Run it three times, using department_ids 10, 20 and 30. What happens and why?

```
DECLARE
  v_employee_id  employees.employee_id%TYPE;
  v_last_name    employees.last_name%TYPE;
BEGIN
  SELECT employee_id, last_name INTO v_employee_id, v_last_name
    FROM employees
   WHERE department_id = <enter a value>;
  DBMS_OUTPUT.PUT_LINE('The SELECT was successful');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An exception has occurred');
END;
```

Using department_id = 10 is successful because there is exactly one employee in that department, therefore the SELECT returned exactly one row.

Using department_id 20 failed because there is more than one employee in that department.

Using department_id 30 failed because there are no employees in that department.

7. Modify your code from question 6 to add two more exception handlers to trap the possible exceptions individually. Use NO_DATA_FOUND and TOO_MANY_ROWS. Re-run the block three times, using 10, 20 and 30 as before. Observe the message displayed in each case.

```
DECLARE
  v_employee_id  employees.employee_id%TYPE;
  v_last_name    employees.last_name%TYPE;
BEGIN
  SELECT employee_id, last_name INTO v_employee_id, v_last_name
    FROM employees
   WHERE department_id = <enter a value>;
  DBMS_OUTPUT.PUT_LINE('The SELECT was successful');
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No rows were selected');
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('More than one row was selected');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An exception has occurred');
END;
```

SECTION 6 LESSON 2 – Trapping Oracle Server Exceptions

Terminology

1. **_Predefined Oracle Server Errors_** Each of these has a predefined name. For example, if the error ORA-01403 occurs when no rows are retrieved from the database in a SELECT statement, then PL/SQL raises the predefined exception-name NO_DATA_FOUND.
2. **_PRAGMA EXCEPTION_INIT_** Tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any Oracle Server exception by name and to write a specific handler for it.
3. **_SQLERRM_** Returns character data containing the message associated with the error number
4. **_Non-predefined Oracle Server Errors_** Each of these has a standard Oracle error number (ORA-nnnnn) and error message, but not a predefined name. We declare our own names for these so that we can reference these names in the exception section.
5. **_SQLCODE_** Returns the numeric value for the error code (You can assign it to a NUMBER variable.)

Try It / Solve It

1. What are the three types of exceptions that can be handled in a PL/SQL block?
 - A. predefined Oracle errors
 - B. non-predefined Oracle errors
 - C. user-defined errors
2. What is the difference in how each of these three types of exception is handled in the PL/SQL block?
 - A. predefined: Oracle has assigned a name to about 20 of the most common errors. We reference the predefined names directly in the exception section of the block;
 - B. non-predefined: these errors do not have names and so a name must be defined in the declare section and associated with an Oracle error number; the name is then used in the exception section;
 - C. user defined: these are unknown to Oracle and must be named in the declare section, raised explicitly in the executable section and then the name used in the exception section

3. Enter and run the following PL/SQL block. Look at the output and answer the following questions:

```
DECLARE
  v_number    NUMBER(6,2) := 100;
  v_region_id wf_world_regions.region_id%TYPE;
  v_region_name wf_world_regions.region_name%TYPE;
BEGIN
  SELECT region_id, region_name INTO v_region_id, v_region_name
    FROM wf_world_regions
   WHERE region_id = 1;
  DBMS_OUTPUT.PUT_LINE('Region: ' || v_region_id ||
                        ' is: ' || v_region_name);
  v_number := v_number / 0;
END;
```

A. What error message is displayed and why ?

Because region_id = 1 does not exist, the predefined exception NO_DATA_FOUND is raised but is not trapped within the block. The exception is propagated back to the calling environment (Application Express) which displays ORA-01403: no data found

B. Modify the block to handle this exception and re-run your code. Now what happens and why?

```
DECLARE
  v_number    NUMBER(6,2) := 100;
  v_region_id wf_world_regions.region_id%TYPE;
  v_region_name wf_world_regions.region_name%TYPE;
BEGIN
  SELECT region_id, region_name INTO v_region_id, v_region_name
    FROM wf_world_regions
   WHERE region_id = 1;
  DBMS_OUTPUT.PUT_LINE('Region: ' || v_region_id ||
                        ' is: ' || v_region_name);
  v_number := v_number / 0;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Select returned no rows');
END;
```

Now the exception is successfully trapped and the corresponding exception handler is executed.

C. Modify the block again to change the WHERE clause to region_id = 29. Re-run the block. Now what happens and why?

Now the SELECT executes successfully because region_id 29 exists. Block execution continues, but the next statement attempts to divide 100 by zero and raises an unhandled ORA-01476 exception.

D. Modify the block again to handle the latest exception and re-run your code.

```
DECLARE
  v_number    NUMBER(6,2) := 100;
  v_region_id wf_world_regions.region_id%TYPE;
  v_region_name wf_world_regions.region_name%TYPE;
BEGIN
  SELECT region_id, region_name INTO v_region_id, v_region_name
    FROM wf_world_regions
    WHERE region_id = 29;
  DBMS_OUTPUT.PUT_LINE('Region: ' || v_region_id ||
    ' is: ' || v_region_name);
  v_number := v_number / 0;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Select returned no rows');
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Attempt to divide by zero');
END;
```

4. Enter and run the following PL/SQL block. Look at the output and answer the following questions:

```
DECLARE
    CURSOR regions_curs IS
        SELECT * FROM wf_world_regions
            WHERE region_id < 20
            ORDER BY region_id;
    regions_rec regions_curs%ROWTYPE;
    v_count NUMBER(6);
BEGIN
    LOOP
        FETCH regions_curs INTO regions_rec;
        EXIT WHEN regions_curs%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Region: ' || regions_rec.region_id ||
                               ' Name: ' || regions_rec.region_name);
    END LOOP;
    CLOSE regions_curs;
    SELECT COUNT(*) INTO v_count
        FROM wf_world_regions
            WHERE region_id = 1;
    DBMS_OUTPUT.PUT_LINE('The number of regions is: ' || v_count);
END;
```

A. What happens and why ?

The first FETCH fails because the cursor has not been opened. The predefined exception INVALID_CURSOR is raised but is not trapped within the block.

B. Modify the block to handle the exception and re-run your code.

Add the following exception section:

```
EXCEPTION
    WHEN INVALID_CURSOR THEN
        DBMS_OUTPUT.PUT_LINE('Attempt to fetch from an unopened
                               cursor ');
```

- C. Modify the block again to add an OPEN statement for the cursor, and re-run your code. Now what happens and why? Remember that region_id = 1 does not exist.

```
DECLARE
  CURSOR regions_curs IS
    SELECT * FROM wf_world_regions
      WHERE region_id < 20
      ORDER BY region_id;
  regions_rec regions_curs%ROWTYPE;
  v_count NUMBER(6);
BEGIN
  OPEN regions_curs;
  LOOP
    FETCH regions_curs INTO regions_rec;
    EXIT WHEN regions_curs%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Region: ' || regions_rec.region_id ||
      ' Name: ' || regions_rec.region_name);
  END LOOP;
  CLOSE regions_curs;
  SELECT COUNT(*) INTO v_count
    FROM wf_world_regions
    WHERE region_id = 1;
  DBMS_OUTPUT.PUT_LINE('The number of regions is: ' || v_count);
EXCEPTION
  WHEN INVALID_CURSOR THEN
    DBMS_OUTPUT.PUT_LINE('Attempt to fetch from an unopened
      cursor');
END;
```

The cursor loop now works fine, so execution continues to the **SELECT COUNT(*)** ... Some students may expect this to fail because region_id 1 does not exist. But a **SELECT COUNT(*)** (unlike **SELECT * ...**) always returns exactly one row, in this case with a value of 0.

5. Oracle Server Errors:

- A. Add an exception handler to the following code to trap the following predefined Oracle Server errors: NO_DATA_FOUND, TOO_MANY_ROWS and DUP_VAL_ON_INDEX.

```
DECLARE
  v_language_id  wf_languages.language_id%TYPE;
  v_language_name wf_languages.language_name%TYPE;
BEGIN
  SELECT language_id, language_name
    INTO v_language_id, v_language_name
    FROM wf_languages
    WHERE LOWER(language_name) LIKE '<substring%>'; -- for example 'ab%'
  INSERT INTO wf_languages(language_id, language_name)
    VALUES(80, null);
END;
```

Add the following exception section:

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No rows were found');
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('More than one row was found');
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('This language id already exists');
```

- B. Test your block twice using each of the following language substrings: ba, ce. There are several language_names beginning with “Ba” but none beginning with “Ce”.
- C. Now test your block a third time using substring: al. There is exactly one language_name beginning with “Al”. Note that language_id 80 (Arabic) already exists. Explain the output.

The SELECT succeeds but the INSERT fails with an ORA-01400 exception because language_name cannot be null.

- D. Now (keeping the substring as “al”) add a non_predefined exception handler to trap the ORA-01400 exception. Name your exception e_null_not_allowed. Rerun the code and observe the results.

Students’ code should now look like this:

```
DECLARE
  v_language_id  wf_languages.language_id%TYPE;
  v_language_name wf_languages.language_name%TYPE;
  e_null_not_allowed EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_null_not_allowed, -01400);
BEGIN
  SELECT language_id, language_name
     INTO v_language_id, v_language_name
     FROM wf_languages
     WHERE lower(language_name) LIKE 'al%';
  INSERT INTO wf_languages(language_id, language_name)
     VALUES(80, null);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
     DBMS_OUTPUT.PUT_LINE('No rows were found');
  WHEN TOO_MANY_ROWS THEN
     DBMS_OUTPUT.PUT_LINE('More than one row was found');
  WHEN DUP_VAL_ON_INDEX THEN
     DBMS_OUTPUT.PUT_LINE('This language id already exists');
  WHEN e_null_not_allowed THEN
     DBMS_OUTPUT.PUT_LINE('Language name cannot be null');
END;
```


Extension Exercise

1. In preparation for this exercise, run the following SQL statement to create an error-logging table:

```
CREATE TABLE error_log (who          VARCHAR2(30),
                        when          DATE,
                        error_code     NUMBER(6),
                        error_message  VARCHAR2(255));
```

Modify your PL/SQL block from question 5 to remove the four explicit exception handlers, replacing them with a single WHEN OTHERS handler. The handler should INSERT a row into the error_log table each time an exception is raised and handled. The row should consist of the Oracle username (who), when the error was raised (when) and the SQLCODE and SQLERRM of the exception. Test your block several times, with different data values to raise each of the four kinds of exception handled in the block. Finally, SELECT from the error-logging table to check that the rows have been inserted.

```
DECLARE
  v_language_id  wf_languages.language_id%TYPE;
  v_language_name wf_languages.language_name%TYPE;
  v_sqlcode      NUMBER(5);
  v_sqlerrm      VARCHAR2(255);
BEGIN
  SELECT language_id, language_name
    INTO v_language_id, v_language_name
    FROM wf_languages
    WHERE lower(language_name) LIKE '<substring%>';
  INSERT INTO wf_languages(language_id, language_name)
    VALUES(80, null);
EXCEPTION
  WHEN OTHERS THEN
    v_sqlcode := SQLCODE;
    v_sqlerrm := SQLERRM;
    INSERT INTO error_log(who, when, error_code, error_message)
      VALUES(USER, SYSDATE, v_sqlcode, v_sqlerrm);
END;

SELECT * FROM error_log;
```

SECTION 6 LESSON 3 – Trapping User-Defined Exceptions

Terminology

1. **RAISE_APPLICATION_ERROR** A procedure used to return user-defined error messages from stored subprograms.
2. **RAISE** Use this statement to raise a named exception.
3. **user-defined errors** These errors are not automatically raised by the Oracle Server, but are defined by the programmer and are specific to the programmer's code.

Try It / Solve It

All the questions in this exercise uses a copy of the employees table. Create this copy by running the following SQL statement:

```
CREATE TABLE excep_emps AS SELECT * FROM employees;
```

1. Create a PL/SQL block that updates the salary of every employee to a new value of 10000 in a chosen department. Include a user-defined exception handler that handles the condition where no rows are updated and displays a custom message. Also include an exception handler that will trap any other possible error condition and display the corresponding SQLCODE and SQLERRM. Test your code three times, using department_ids 20, 30 and 40.

DECLARE

e_no_rows_updated EXCEPTION;

BEGIN

UPDATE excep_emps

SET salary = 10000

WHERE department_id = <enter a department_id>;

IF SQL%NOTFOUND THEN -- or we could say: IF SQL%ROWCOUNT = 0

RAISE e_no_rows_updated;

END IF;

EXCEPTION

WHEN e_no_rows_updated THEN

**DBMS_OUTPUT.PUT_LINE ('There are no employees in that
department.');**

WHEN OTHERS THEN

**DBMS_OUTPUT.PUT_LINE('An error has occurred:'
||SQLCODE || '-' || SQLERRM);**

END;

2. Modify your code from question 2 to handle the condition where no rows are updated using **RAISE_APPLICATION_ERROR** procedure in the exception section. Use an error number of -20202. Test your code again using department_id 40 and check that the -20202 error is displayed.

DECLARE

```

e_no_rows_updated EXCEPTION;
BEGIN
  UPDATE excep_emps
    SET salary = 10000
    WHERE department_id = <enter a dept_id>;
  IF SQL%NOTFOUND THEN
    RAISE e_no_rows_updated;
  END IF;
EXCEPTION
  WHEN e_no_rows_updated THEN
    RAISE_APPLICATION_ERROR(-20202,
      'There are no employees in that department');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An error has occurred: '
      ||SQLCODE || '-' || SQLERRM);
END;

```

3. Modify your code from question 3 to use RAISE_APPLICATION_ERROR in the executable section instead of the exception section. Test your code again using department_id 40.

```

BEGIN
  UPDATE excep_emps
    SET salary = 10000
    WHERE department_id = <enter a dept_id>;
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20202,
      'There are no employees in that department');
  END IF;
END;

```

Observe that because there is now no exception handler for e_no_rows_updated, we do not need to declare it anymore.

4. Before starting this question, disable Autocommit in Application Express.

A. Enter and run the following PL/SQL block using department_id = 40, and explain the output.

```
DECLARE
  v_dept_id  excep_emps.department_id%TYPE;
  v_count    NUMBER;
BEGIN
  v_dept_id := 40;
  SELECT COUNT(*) INTO v_count
    FROM excep_emps
       WHERE department_id = v_dept_id;
  DBMS_OUTPUT.PUT_LINE('There are ' || v_count || ' employees');
  DELETE FROM excep_emps
     WHERE department_id = v_dept_id;
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||
    ' employees were deleted');
  ROLLBACK;
END;
```

Both the SELECT and the DELETE execute successfully (no exceptions are raised) although there are no employees in department 40.

- B. Modify your block to include two user_defined exception handlers, one to test whether SELECT returns a value of 0, the other to test if no rows were DELETED. Declare the exceptions and RAISE them explicitly before trapping them in the exception section. Do NOT use RAISE_APPLICATION_ERROR. Test your modified block using department_id 40.

```
DECLARE
v_dept_id  excep_emps.department_id%TYPE;
v_count    NUMBER;
e_no_emps_in_dept EXCEPTION;
e_no_rows_deleted EXCEPTION;

BEGIN
v_dept_id := 40;
SELECT COUNT(*) INTO v_count
  FROM excep_emps
     WHERE department_id = v_dept_id;
IF v_count = 0 THEN
  RAISE e_no_emps_in_dept;
END IF;
DBMS_OUTPUT.PUT_LINE('There are ' || v_count || ' employees');
DELETE FROM excep_emps
  WHERE department_id = v_dept_id;
IF SQL%NOTFOUND THEN -- or IF SQL%ROWCOUNT = 0 THEN
  RAISE e_no_rows_deleted;
END IF;
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||
  ' employees were deleted');
ROLLBACK;
EXCEPTION
  WHEN e_no_emps_in_dept THEN
    DBMS_OUTPUT.PUT_LINE('This department has no employees');
  WHEN e_no_rows_deleted THEN
    DBMS_OUTPUT.PUT_LINE('No employees were deleted');
END;
```

- C. Modify your block again to use `RAISE_APPLICATION_ERROR` in the executable section. Use error numbers –20203 and –20204. Test your modified block using `department_id` 40.

```
DECLARE
  v_dept_id  excep_emps.department_id%TYPE;
  v_count    NUMBER;

BEGIN
  v_dept_id := 40;
  SELECT COUNT(*) INTO v_count
    FROM excep_emps
    WHERE department_id = v_dept_id;
  IF v_count = 0 THEN
    RAISE_APPLICATION_ERROR(-20203,
      'This department has no employees');
  END IF;
  DBMS_OUTPUT.PUT_LINE('There are ' || v_count || ' employees');
  DELETE FROM excep_emps
    WHERE department_id = v_dept_id;
  IF SQL%NOTFOUND THEN -- or IF SQL%ROWCOUNT = 0 THEN
    RAISE_APPLICATION_ERROR(-20204,
      'No employees were deleted');
  END IF;
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||
    ' employees were deleted');
  ROLLBACK;
END;
```

SECTION 6 LESSON 4 – Recognizing the Scope of Variables

Terminology

1. **Visibility** The portion of the program where the variable can be accessed without using a qualifier.
2. **Qualifier** A label given to a block.
3. **Scope** The portion of a program in which the variable is declared and is accessible.

Try It / Solve It

1. In your own words, explain the difference between the scope of a variable and the visibility of a variable. Give an example where a variable **is** in scope but **not** visible.

The scope of a variable is the set of blocks in which it can be referenced and used. The visibility of a variable is the set of blocks in which it can be referenced and used without needing to be qualified by a block label.

Example: in this code, the scope of the outer block's v_myvar is both blocks, but it is not visible in the inner block.

```
DECLARE
  v_myvar  NUMBER;
BEGIN
  DECLARE
    v_myvar  NUMBER;
  BEGIN
    ...
  END;
  ...
END;
```

2. Enter and run the following PL/SQL code.

```
DECLARE
  v_last_name employees.last_name%TYPE;
BEGIN
  SELECT last_name INTO v_last_name
    FROM employees WHERE employee_id = 100;
  /* This employee's last name is King */
  DECLARE
    v_last_name employees.last_name%TYPE;
  BEGIN
    SELECT last_name INTO v_last_name
      FROM employees WHERE employee_id = 107;
    /* This employee's last name is Lorentz */
    DBMS_OUTPUT.PUT_LINE(v_last_name);
  END;
  DBMS_OUTPUT.PUT_LINE(v_last_name);
END;
```

- A. What output is displayed, and why? Save your code.

Output is: **Lorentz**
 King

v_last_name is declared in both blocks, and each DBMS_OUTPUT.PUT_LINE uses the most locally scoped occurrence of v_last_name.

- B. Modify the code by using a block label so that the inner block displays both employees' last names while the outer block displays nothing. Execute and save your code.

```
<<outer>>
DECLARE
  v_last_name  employees.last_name%TYPE;
BEGIN
  SELECT last_name INTO v_last_name
    FROM employees WHERE employee_id = 100;
  /* This employee's last name is King */
  DECLARE
    v_last_name  employees.last_name%TYPE;
  BEGIN
    SELECT last_name INTO v_last_name
      FROM employees WHERE employee_id = 107;
    /* This employee's last name is Lorentz */
    DBMS_OUTPUT.PUT_LINE(v_last_name);
    DBMS_OUTPUT.PUT_LINE(outer.v_last_name);
  END;
END;
```

- C. Modify the code again so that this time, the outer block displays both employees' last names while the inner block displays nothing. Execute your code.

Since we cannot reference the inner block's variable in the outer block, we must declare a second variable in the outer block and assign the value of the inner block's variable to it.

```
DECLARE
  v_last_name      employees.last_name%TYPE;
  v_inner_last_name employees.last_name%TYPE;
BEGIN
  SELECT last_name INTO v_last_name
    FROM employees WHERE employee_id = 100;
  /* This employee's last name is King */
  DECLARE
    v_last_name  employees.last_name%TYPE;
  BEGIN
    SELECT last_name INTO v_last_name
      FROM employees WHERE employee_id = 107;
    /* This employee's last name is Lorentz */
    v_inner_last_name := v_last_name; -- assignment
  END;
  DBMS_OUTPUT.PUT_LINE(v_inner_last_name);
  DBMS_OUTPUT.PUT_LINE(v_last_name);
END;
```

3. Enter and execute the following block, which uses a cursor with a parameter to fetch and display the country name and area of countries in Central America. You should see eight countries displayed. Save your code.

```
DECLARE
v_region_id    wf_world_regions.region_id%TYPE;
CURSOR country_curs
    (p_region_id wf_world_regions.region_id%TYPE) IS
    SELECT country_name, area
    FROM      wf_countries
    WHERE     region_id = p_region_id
    ORDER BY country_name;
country_rec    country_curs%ROWTYPE;

BEGIN
    SELECT region_id INTO v_region_id
    FROM wf_world_regions
    WHERE region_name = 'Central America';
    OPEN country_curs(v_region_id);
    LOOP
        FETCH country_curs INTO country_rec;
        EXIT WHEN country_curs%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (country_rec.country_name ||
                               ' ' || country_rec.area);
    END LOOP;
    CLOSE country_curs;
END;
```

- A. Modify the code to place all the cursor references (DECLARE, OPEN, FETCH in a loop and CLOSE) in an inner nested block. Leave the declaration of v_region_id and the single-row SELECT in the outer block. Execute your code (you should see the same output as in step a). Save your code.

```
DECLARE -- outer block
v_region_id wf_world_regions.region_id%TYPE;
BEGIN
SELECT region_id INTO v_region_id
FROM wf_world_regions
WHERE region_name = 'Central America';
DECLARE -- inner block
CURSOR country_curs
(p_region_id wf_world_regions.region_id%TYPE) IS
SELECT country_name, area
FROM wf_countries
WHERE region_id = p_region_id
ORDER BY country_name;
country_rec country_curs%ROWTYPE;
BEGIN
OPEN country_curs(v_region_id);
LOOP
FETCH country_curs INTO country_rec;
EXIT WHEN country_curs%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (country_rec.country_name ||
' ' || country_rec.area);
END LOOP;
CLOSE country_curs;
END;
END;
```

- B. Now modify the code again by moving the CLOSE *cursor-name* statement to the outer block. Execute your code. What happens and why ?

A PLS-00201 “Identifier COUNTRY_CURS must be declared” error is returned, because the cursor was declared in the inner block and cannot be referenced in the outer block.

4. Enter and run the following code twice, using country_ids 5 (which does not exist) and 672 (Antarctica, which does exist but has no currency).

```
DECLARE
  v_country_name wf_countries.country_name%TYPE;
  v_currency_code wf_countries.currency_code%TYPE;
BEGIN
  DECLARE
    e_no_currency EXCEPTION;
  BEGIN
    SELECT country_name, currency_code
      INTO v_country_name, v_currency_code
    FROM wf_countries
      WHERE country_id = 5; -- repeat with 672
    IF v_currency_code = 'NONE' THEN
      RAISE e_no_currency;
    END IF;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('This country does not exist');
    WHEN e_no_currency THEN
      DBMS_OUTPUT.PUT_LINE('This country exists but has no currency');
  END;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Another type of error occurred');
END;
```

- A. Explain the output. Save your code.

Country_id 5 raises NO_DATA_FOUND and 'This country does not exist' is displayed. Country_id 672 raises e_no_currency and 'This country exists but has no currency' is displayed.

- B. Modify the code to move the two exception handlers to the outer block. Leave the declaration of `e_no_currency` in the inner block. Execute twice, again using `country_ids` 5 and 672. Now what happens and why? Save your code.

```
DECLARE
  v_country_name  wf_countries.country_name%TYPE;
  v_currency_code wf_countries.currency_code%TYPE;
BEGIN
  DECLARE
    e_no_currency EXCEPTION;
  BEGIN
    SELECT country_name, currency_code
      INTO v_country_name, v_currency_code
    FROM wf_countries
      WHERE country_id = 5; -- repeat with 672
    IF v_currency_code = 'NONE' THEN
      RAISE e_no_currency;
    END IF;
  END;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('This country does not exist');
  WHEN e_no_currency THEN
    DBMS_OUTPUT.PUT_LINE('This country exists but has no currency');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Another type of error occurred');
END;
```

In both executions, a PLS-00201 “Identifier E_NO_CURRENCY must be declared” error occurs, because `e_no_currency` was declared in the inner block and is therefore out of scope in the outer block. We would expect this for `country_id` 672, but what about `country_id` 5, which should raise a `NO_DATA_FOUND`? PLS-00201 is a compile-time error. This shows that PL/SQL checks the scope of variables when compiling the code, not when executing it. If invalid scope is detected, the code is not executed at all.

C. Modify the code again to move the declaration of `e_no_currency` to the outer block. Re-execute again using `country_ids` 5 and 672. Now what happens and why?

```
DECLARE
  v_country_name  wf_countries.country_name%TYPE;
  v_currency_code wf_countries.currency_code%TYPE;
  e_no_currency   EXCEPTION;
BEGIN
  BEGIN
    SELECT country_name, currency_code
      INTO v_country_name, v_currency_code
      FROM wf_countries
      WHERE country_id = 5; -- repeat with 672
    IF v_currency_code = 'NONE' THEN
      RAISE e_no_currency;
    END IF;
  END;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('This country does not exist');
  WHEN e_no_currency THEN
    DBMS_OUTPUT.PUT_LINE('This country exists but has no currency');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Another type of error occurred');
END;
```

Now both executions are successful, because `e_no_currency` is in scope in the outer block.