Chapter 09

Advanced Exception Handling in PL/SQL

Mayko Freitas da Silva

Introduction

Imagine you're building a house. You've got your blueprints, your tools, and your materials. But what happens when something unexpected occurs? Maybe a storm rolls in, or you discover an issue with the foundation. In construction, you need contingency plans. In PL/SQL programming, these contingency plans are called exception handling.

In this chapter, you're going to take your exception handling skills to the next level. You'll learn about three key concepts that will help you write more robust and reliable PL/SQL code:

1. Exception Scope: You'll discover how exceptions behave in different parts of your code, much like understanding how different rooms in a house are affected by various issues.
2. User-Defined Exceptions: You'll learn to create your own custom exceptions, giving you the power to handle specific situations unique to your programs.
3. Exception Propagation: You'll explore how exceptions move through your code, similar to how a problem in one part of a house can affect other areas.

By the end of this chapter, you'll be equipped with advanced techniques to handle errors gracefully, making your PL/SQL programs more resilient and user-friendly. Whether you're dealing with unexpected user inputs, database issues, or complex business logic, you'll have the tools to ensure your code responds appropriately to any situation.

Remember, effective exception handling is not just about preventing crashes. It's about creating programs that can adapt to real-world scenarios, provide meaningful feedback, and maintain data integrity even when things don't go as planned.

So, are you ready to become an exception handling expert? Let's dive in and explore these advanced concepts together.

Lab 9.1: Understanding Exception Scope

After this lab, you will be able to:

- Define exception scope
- Apply scope rules for exceptions
- Handle exceptions in nested blocks

What is Exception Scope?

You're already familiar with the term scope—for example, the scope of a variable. Even though variables and exceptions serve different purposes, the same scope rules apply to them. These rules are best illustrated by means of an example.

For Example: ch09_1a.sql

```
DECLARE
  v_employee_id NUMBER := &sv_employee_id;
  v_name       VARCHAR2(50);
BEGIN
  SELECT first_name || ' ' || last_name
    INTO v_name
    FROM employees
   WHERE employee_id = v_employee_id;
  DBMS_OUTPUT.PUT_LINE('Employee name is ' || v_name);
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE('There is no such employee');
END;
```

In this example, the employee's name is displayed on the screen for a given value of employee ID provided at runtime. If there is no record in the EMPLOYEES table corresponding to the value of v_employee_id, the exception NO_DATA_FOUND is raised.

Therefore, you can say that the exception NO_DATA_FOUND covers this block or that this block is the scope of this exception. In other words, the scope of an exception is the portion of the block that is covered by this exception.

Now, you can expand on that understanding (newly added statements are shown in bold).

For Example: ch09_1b.sql

```
<<outer_block>>
DECLARE
  v_employee_id NUMBER := &sv_employee_id;
  v_name       VARCHAR2(50);
  v_dept_count  NUMBER(2);
BEGIN
  SELECT first_name || ' ' || last_name
    INTO v_name
    FROM employees
   WHERE employee_id = v_employee_id;
  DBMS_OUTPUT.PUT_LINE('Employee name is ' || v_name);

  <<inner_block>>
  BEGIN
    SELECT COUNT(*)
      INTO v_dept_count
      FROM departments
     WHERE manager_id = v_employee_id;
    DBMS_OUTPUT.PUT_LINE('Employee manages ' || v_dept_count || ' department(s)');
  EXCEPTION
    WHEN VALUE_ERROR OR INVALID_NUMBER
    THEN
      DBMS_OUTPUT.PUT_LINE('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE('There is no such employee');
END;
```

The new version of the example includes an inner block. This block has a structure similar to the outer block; that is, it has a SELECT INTO statement and an exception section to handle errors. When a VALUE_ERROR or INVALID_NUMBER error occurs in the inner block, the exception is raised.

Notice that the exceptions VALUE_ERROR and INVALID_NUMBER have been defined for the inner block only. Therefore, they can be handled only if they are raised in the inner block. If one of these errors occurs in the outer block, the program will be unable to terminate successfully.

In contrast, the exception NO_DATA_FOUND has been defined in the outer block; therefore, it is global to the inner block. However, this version of the example never raises the exception NO_DATA_FOUND in the inner block. Why do you think this is the case?

Did You Know? If you define an exception in a block, it is local to that block. However, it is global to any blocks enclosed by that block. In other words, in the case of nested blocks, any exception defined in the outer block becomes global to its inner blocks.

Note what happens when the example is changed so that the exception NO_DATA_FOUND can be raised by the inner block (all changes are shown in bold).

For Example:

```
<<outer_block>>
DECLARE
  v_employee_id NUMBER := &sv_employee_id;
  v_name       VARCHAR2(50);
  v_is_manager  CHAR(1);
BEGIN
  SELECT first_name || ' ' || last_name
    INTO v_name
    FROM employees
   WHERE employee_id = v_employee_id;
  DBMS_OUTPUT.PUT_LINE('Employee name is ' || v_name);

  <<inner_block>>
  BEGIN
    SELECT 'Y'
      INTO v_is_manager
      FROM departments
     WHERE manager_id = v_employee_id
       AND ROWNUM = 1;
    DBMS_OUTPUT.PUT_LINE('Employee is a manager');
  EXCEPTION
    WHEN VALUE_ERROR OR INVALID_NUMBER
    THEN
      DBMS_OUTPUT.PUT_LINE('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE('There is no such employee');
END;
```

The new version of the example has a different SELECT INTO statement. To answer the question posed earlier, the exception NO_DATA_FOUND can be raised by the inner block because the SELECT INTO statement does not contain a group function, COUNT(). This function always returns a result, so when no rows are returned by the SELECT INTO statement, the value returned by the COUNT(*) equals zero.

Now, consider the output produced by this example when a value of 100 is provided for the employee ID:

```
Employee name is Steven King
There is no such employee
```

You have probably noticed that this example produces only a partial output. Even though you are able to see the employee's name, the error message is displayed, indicating that this employee does not exist. This error message is displayed because the exception NO_DATA_FOUND is raised in the inner block.

The SELECT INTO statement of the outer block returns the employee's name, which is then displayed on the screen by the first DBMS_OUTPUT.PUT_LINE statement. Next, control passes to the inner block. The SELECT INTO statement of the inner block does not return any rows. As a result, an error occurs and the NO_DATA_FOUND exception is raised.

Next, PL/SQL tries to find a handler for the NO_DATA_FOUND exception in the inner block. Because there is no such handler in the inner block, control is transferred to the exception section of the outer block. The exception section of the outer block contains the handler for the exception NO_DATA_FOUND. Consequently, this handler executes, and the message "There is no such employee" is displayed on the screen. The process, which is called exception propagation, is discussed in detail in Lab 9.3.

Be aware that this example has been provided for illustrative purposes only. In its current version, it is not very useful. The SELECT INTO statement of the inner block is prone to another exception, TOO_MANY_ROWS, which is not handled by this example. In addition, the error message "There is no such employee" is not very descriptive when the exception NO_DATA_FOUND is raised by the inner block.

Lab 9.2: Creating User-Defined Exceptions

After this lab, you will be able to:

- Declare your own custom exceptions
- Raise user-defined exceptions
- Handle user-defined exceptions in your code

Often in your programs, you may need to handle problems that are specific to the program you write. For example, suppose your program asks a user to enter a value for the employee ID. This value is then assigned to the variable v_employee_id, which is used later in the program. Generally, you want a positive number for an ID. By mistake, however, the user enters a negative number. However, no error has occurred because the variable v_employee_id has been defined as a number, and the user has supplied a legitimate numeric value. Therefore, you may want to implement your own exception to handle this situation.

This type of exception is called a user-defined exception because it is defined by the programmer. As a result, before such an exception can be used, it must be declared. A user-defined exception is declared in the declarative part of a PL/SQL block, as shown in Listing 9.1.

Listing 9.1 User-Defined Exception Declaration

```
DECLARE
  exception_name EXCEPTION;
```

Notice that this declaration looks similar to a variable declaration. That is, you specify an exception name followed by the keyword EXCEPTION. Consider the following code fragment.

For Example:

```
DECLARE
e_invalid_id EXCEPTION;
```

In this code fragment, the name of the exception is prefixed by the letter e. This is not a required syntax; rather, it allows you to differentiate between variable names and exception names.

After an exception has been declared, the executable statements associated with that exception are specified in the exception-handling section of the block. The format of the exception-handling section is the same as for built-in exceptions. Consider the following code fragment.

For Example:

```
DECLARE
  e_invalid_id EXCEPTION;
BEGIN
  ...
EXCEPTION
  WHEN e_invalid_id
  THEN
    DBMS_OUTPUT.PUT_LINE('An ID cannot be negative');
END;
```

You already know that built-in exceptions are raised implicitly. In other words, when a certain error occurs, a built-in exception associated with this error is raised. Of course, you are assuming that you have included this exception in the exception-handling section of your program. For example, a TOO_MANY_ROWS exception is raised when a SELECT INTO statement returns multiple rows.

A user-defined exception must be raised explicitly. In other words, you need to specify in your program under which circumstances an exception must be raised, as shown in Listing 9.2.

Listing 9.2 Raising a User-Defined Exception

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  ...
  IF CONDITION
  THEN
    RAISE exception_name;
  END IF;
  ...
EXCEPTION
  WHEN exception_name
  THEN
    ERROR-PROCESSING STATEMENTS;
END;
```

In this structure, the circumstances under which a user-defined exception must be raised are determined with the help of the IF statement. If CONDITION evaluates to TRUE, a user-defined exception is raised with the help of the RAISE statement. If CONDITION evaluates to FALSE, the program proceeds with its normal execution. In other words, the statements following the IF statement are executed. Note that any form of the IF statement can be used to check when a user-defined exception must be raised.

In the next example, which is based on the code fragments provided earlier in this lab, you will see that the exception e_invalid_id is raised when the user enters a negative number for the variable v_employee_id.

For Example:

```
DECLARE
  v_employee_id    employees.employee_id%TYPE := &sv_employee_id;
  v_total_jobs     NUMBER;
  e_invalid_id     EXCEPTION;
BEGIN
  IF v_employee_id < 0
  THEN
    RAISE e_invalid_id;
  END IF;

  SELECT COUNT(*)
   INTO v_total_jobs
   FROM job_history
   WHERE employee_id = v_employee_id;

  DBMS_OUTPUT.PUT_LINE('The employee has held ' ||
                v_total_jobs || ' job(s) in the past');
  DBMS_OUTPUT.PUT_LINE('No exception has been raised');
EXCEPTION
  WHEN e_invalid_id
  THEN
    DBMS_OUTPUT.PUT_LINE('An ID cannot be negative');
END;
```

In this example, the exception e_invalid_id is raised with the help of the IF statement. When a value is supplied for the variable v_employee_id, the sign of this numeric value is checked. If the value is less than zero, the IF statement evaluates to TRUE, and the exception e_invalid_id is raised. In turn, control passes to the exception-handling section of the block. Next, statements associated with this exception are executed. In this case, the message "An ID cannot be negative" is displayed on the screen. If the value entered for the v_employee_id is positive, the IF statement yields FALSE and the rest of the statements in the body of the block are executed.

Consider executing this example for two values of v_employee_id, 102 and –102. The first run of the example (the employee ID is 102) produces the following output:

```
The employee has held 1 job(s) in the past
No exception has been raised
```

For this run, the user provides a positive value for the variable v_employee_id. As a result, the IF statement evaluates to FALSE, and the SELECT INTO statement determines how many records are in the JOB_HISTORY table for the given employee ID. Next, the messages "The employee has held 1 job(s) in the past" and "No exception has been raised" are displayed on the screen. At this point, the body of the PL/SQL block has executed to completion.

A second run of the example (the employee ID is –102) produces the following output:

```
An ID cannot be negative
```

For this run, the user entered a negative value for the variable v_employee_id. The IF statement evaluates to TRUE, and the exception e_invalid_id is raised. As a result, control of the execution passes to the exception-handling section of the block, and the message "An ID cannot be negative" is displayed on the screen.

Watch Out. The RAISE statement should be used in conjunction with an IF statement. Otherwise, control of the execution will be transferred to the exception-handling section of the block for every single execution. Consider the following example:

```
DECLARE
  e_test_exception EXCEPTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Exception has not been raised');
  RAISE e_test_exception;
  DBMS_OUTPUT.PUT_LINE('Exception has been raised');
EXCEPTION
  WHEN e_test_exception
  THEN
    DBMS_OUTPUT.PUT_LINE('An error has occurred');
END;
```

Every time this example is run, the following output is produced:

```
Exception has not been raised
An error has occurred
```

Even though no error has occurred, control is transferred to the exception-handling section. It is important for you to check whether the error has occurred before raising the exception associated with that error.

The same scope rules apply to user-defined exceptions as apply to built-in exceptions. An exception declared in the inner block must be raised in the inner block and defined in the exception-handling section of the inner block. Consider the following example.

For Example:

```
<<outer_block>>
BEGIN
  DBMS_OUTPUT.PUT_LINE('Outer block');
  <<inner_block>>
  DECLARE
    e_my_exception EXCEPTION;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inner block');
  EXCEPTION
    WHEN e_my_exception
    THEN
      DBMS_OUTPUT.PUT_LINE('An error has occurred in the inner block');
  END;

  IF 10 > &sv_number
  THEN
    RAISE e_my_exception;
  END IF;
END;
```

In this example, the exception, e_my_exception, has been declared in the inner block. However, you are trying to raise this exception in the outer block. This example causes a syntax error because the exception declared in the inner block ceases to exist after the inner block terminates. As a result, this example produces the following output when a value of 11 is provided at runtime:

```
ORA-06550: line 19, column 13:
PLS-00201: identifier 'E_MY_EXCEPTION' must be declared
ORA-06550: line 19, column 7:
PL/SQL: Statement ignored
```

Notice that the error message

```
PLS-00201: identifier 'E_MY_EXCEPTION' must be declared
```

is the same error message you get when you try to use a variable that has not been declared.

Lab 9.3: Exception Propagation and Re-raising

After this lab, you will be able to:

- Understand how exceptions propagate through different parts of your code
- Re-raise exceptions when needed
- Apply best practices for exception handling in complex scenarios

You already have seen how different types of exceptions are raised when a runtime error occurs in the executable portion of the PL/SQL block. However, a runtime error may also occur in the declaration section of the block or in the exception-handling section of the block. The rules that govern how exceptions are raised in these situations are referred to as exception propagation.

Consider the first case, in which a runtime error occurs in the executable section of the PL/SQL block. This case should be treated as a review because the examples given earlier in this chapter show how an exception is raised when an error occurs in the executable section of the block.

If a specific exception is associated with a particular error, control passes to the exception-handling section of the block. After the statements associated with the exception are executed, control passes to the host environment or to the enclosing block. If there is no exception handler for this error, the exception is propagated to the enclosing block (outer block). The steps just described are then repeated. If no exception handler is found, the execution of the program halts, and control is transferred to the host environment.

Next, consider the second case, in which a runtime error occurs in the declaration section of the block. If there is no outer block, the execution of the program halts, and control passes to the host environment. Consider the following example.

For Example:

```
DECLARE
  v_test_var CHAR(3) := 'ABCDE';
BEGIN
  DBMS_OUTPUT.PUT_LINE('This is a test');
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    DBMS_OUTPUT.PUT_LINE('An error has occurred');
END;
```

When executed, this example produces the following output:

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 2
```

In this example, the assignment statement in the declaration section of the block causes an error. Even though an exception handler for this error exists, the block is not able to execute successfully. Based on this example, you can conclude that when a runtime error occurs in the declaration section of the PL/SQL block, the exception-handling section of this block is not able to catch the error.

Next, consider a modified version of the same example that employs nested PL/SQL blocks (changes are shown in bold).

For Example:

```
<<outer_block>>
BEGIN
  <<inner_block>>
  DECLARE
    v_test_var CHAR(3) := 'ABCDE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('This is a test');
  EXCEPTION
    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
      DBMS_OUTPUT.PUT_LINE('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    DBMS_OUTPUT.PUT_LINE('An error has occurred in the program');
END;
```

When executed, this example produces the following output:

```
An error has occurred in the program
```

In this version of the example, the PL/SQL block is enclosed by another block, and the program is able to complete. In this case, the exception defined in the outer block is raised when the error occurs in the declaration section of the inner block. Therefore, you can conclude that when a runtime error occurs in the declaration section of the inner block, the exception immediately propagates to the enclosing (outer) block.

Finally, consider a third case, in which a runtime error occurs in the exception-handling section of the block. Just as in the previous case, if there is no outer block, the execution of the program halts and control passes to the host environment. Consider the following example.

For Example:

```
DECLARE
  v_test_var CHAR(3) := 'ABC';
BEGIN
  v_test_var := '1234';
  DBMS_OUTPUT.PUT_LINE('v_test_var: ' || v_test_var);
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    v_test_var := 'ABCD';
    DBMS_OUTPUT.PUT_LINE('An error has occurred');
END;
```

When executed, this example produces the following output:

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 9
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 4
```

As you can see, the assignment statement in the executable section of the block causes an error. In turn, control is transferred to the exception-handling section of the block. However, the assignment statement in the exception-handling section of the block raises the same error. As a result, the output of this example displays the same error message twice. The first message is generated by the assignment statement in the executable section of the block, and the second message is generated by the assignment statement of the exception-handling section of this block. Based on this example, you can conclude that when a runtime error occurs in the exception-handling section of the PL/SQL block, the exception-handling section of this block is not able to prevent the error.

Next, consider a modified version of the same example with nested PL/SQL blocks (affected statements are shown in bold).

For Example:

```
<<outer_block>>
BEGIN
  <<inner_block>>
  DECLARE
    v_test_var CHAR(3) := 'ABC';
  BEGIN
    v_test_var := '1234';
    DBMS_OUTPUT.PUT_LINE('v_test_var: ' || v_test_var);
  EXCEPTION
    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
      v_test_var := 'ABCD';
      DBMS_OUTPUT.PUT_LINE('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    DBMS_OUTPUT.PUT_LINE('An error has occurred in the program');
END;
```

When executed, this version produces the following output:

```
An error has occurred in the program
```

In this version of the example, the PL/SQL block is enclosed by another block, and the program is able to complete. In this case, the exception defined in the outer block is raised when the error occurs in the exception-handling section of the inner block. Therefore, you can conclude that when a runtime error occurs in the exception-handling section of the inner block, the exception immediately propagates to the enclosing block.

In the previous two examples, an exception was raised implicitly by a runtime error in the exception-handling section of the block. However, an exception can also be raised explicitly in the exception-handling section of the block by the RAISE statement. Consider the following example.

For Example:

```
<<outer_block>>
DECLARE
  e_exception1 EXCEPTION;
  e_exception2 EXCEPTION;
BEGIN
  <<inner_block>>
  BEGIN
    RAISE e_exception1;
  EXCEPTION
    WHEN e_exception1
    THEN
      RAISE e_exception2;
    WHEN e_exception2
    THEN
      DBMS_OUTPUT.PUT_LINE('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN e_exception2
  THEN
    DBMS_OUTPUT.PUT_LINE('An error has occurred in the program');
END;
```

This example produces the following output:

```
An error has occurred in the program
```

The declaration portion of the block contains declarations of two exceptions: e_exception1 and e_exception2. The exception e_exception1 is raised in the inner block via the RAISE statement. In the exception-handling section of the inner block, the exception e_exception1 tries to raise e_exception2. Even though an exception handler for e_exception2 exists in the inner block, control is still transferred to the outer block. This happens because only one exception can be raised in the exception-handling section of the block. Only after one exception has been handled can another be raised, but two or more exceptions cannot be raised simultaneously.

Essentially, when the exception e_exception2 is raised in the exception-handling section of the inner block, it cannot be handled in the same exception-handling section. Thus, the portion of the code surrounded by rectangular brackets never executes. Instead, control passes to the exception-handling section of the outer block and the message "An error has occurred in the program" is displayed on the screen.

Watch Out. When an exception is raised in a PL/SQL block that does not have an appropriate exception-handling mechanism and is not enclosed by another block, control is transferred to the host environment, and the program is not able to complete successfully.

The following code fragment illustrates this case:

```
DECLARE
  e_exception1 EXCEPTION;
BEGIN
  RAISE e_exception1;
END;
```

When run, this example produces the following output:

```
DECLARE
  e_exception1 EXCEPTION;
BEGIN
  RAISE e_exception1;
END;
```

Note that this behavior applies to built-in exceptions and was also seen in Chapter 8, "Error Handling and Built-in Exceptions."

Re-raising Exceptions

On some occasions, you might want to be able to stop your program if a certain type of error occurs. In other words, you might want to handle an exception in the inner block and then pass it to the outer block. This process is called re-raising an exception. The following example helps to illustrate this point.

For Example:

```
<<outer_block>>
DECLARE
  e_exception EXCEPTION;
BEGIN
  <<inner_block>>
  BEGIN
    RAISE e_exception;
  EXCEPTION
    WHEN e_exception
    THEN
      RAISE;
  END;
EXCEPTION
  WHEN e_exception
  THEN
    DBMS_OUTPUT.PUT_LINE('An error has occurred');
END;
```

In this example, the exception e_exception is first declared in the outer block, then raised in the inner block. As a result, control is transferred to the exception-handling section of the inner block. The RAISE statement in the exception-handling section of the block causes the exception to propagate to the exception-handling section of the outer block. Notice that when the RAISE statement is used in the exception-handling section of the inner block, it is not followed by the exception name.

When run, this example produces the following output:

```
An error has occurred
```

Watch Out. When an exception is re-raised in the block that is not enclosed by any other block, the program is unable to complete successfully. Consider the following example:

```
DECLARE
  e_exception EXCEPTION;
BEGIN
  RAISE e_exception;
EXCEPTION
  WHEN e_exception
  THEN
    RAISE;
END;
```

When run, this example produces the following output:

```
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 7
```

Summary

In this chapter, you learned about exception scope and propagation, and saw how to define and raise your own exceptions. In addition, you learned how to re-raise an exception. These advanced exception handling techniques will help you create more robust and reliable PL/SQL programs, capable of gracefully handling various error scenarios.

Key points to remember:

1. Exception scope determines where an exception can be caught and handled.

2. User-defined exceptions allow you to handle custom error situations in your code.
3. Exception propagation describes how exceptions move through nested blocks when they're not handled locally.
4. Re-raising exceptions allows you to handle an exception in an inner block and then pass it to an outer block for further processing.

By mastering these concepts, you'll be better equipped to write PL/SQL code that can effectively manage errors and unexpected situations, leading to more stable and maintainable applications.

Advanced Exception Handling in PL/SQL