Skip Headers

PL/SQL User's Guide and Reference
Release 2 (9.2)
Part Number A96624-01

Home Book List Contents Index Master Index Feedback

Previous Next

7
Handling PL/SQL Errors

There is nothing more exhilarating than to be shot at without result. --Winston Churchill

Run-time errors arise from design faults, coding mistakes, hardware failures, and many other sources. Although you cannot anticipate all possible errors, you can plan to handle certain kinds of errors meaningful to your PL/SQL program.

With many programming languages, unless you disable error checking, a run-time error such as *stack overflow* or *division by zero* stops normal processing and returns control to the operating system. With PL/SQL, a mechanism called *exception handling* lets you "bulletproof" your program so that it can continue operating in the presence of errors.

This chapter discusses the following topics:

Overview of PL/SQL Error Handling

In PL/SQL, a warning or error condition is called an *exception*. Exceptions can be internally defined (by the run-time system) or user defined. Examples of internally defined exceptions include *division by zero* and *out of memory*. Some common internal exceptions have predefined names, such as `ZERO_DIVIDE` and `STORAGE_ERROR`. The other internal exceptions can be given names.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named `insufficient_funds` to flag overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions *must* be given names.

When an error occurs, an exception is *raised*. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by `RAISE` statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called *exception handlers*. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

In the example below, you calculate and store a price-to-earnings ratio for a company with ticker symbol XYZ. If the company has zero earnings, the predefined exception `ZERO_DIVIDE` is raised. This stops normal execution of the block and transfers control to the exception handlers. The optional `OTHERS` handler catches all exceptions that the block does not name specifically.

```
DECLARE
   pe_ratio NUMBER(3,1);
BEGIN
   SELECT price / earnings INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ';  -- might cause division-by-zero error
   INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
   COMMIT;
EXCEPTION  -- exception handlers begin
   WHEN ZERO_DIVIDE THEN  -- handles 'division by zero' error
      INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
      COMMIT;
   ...
   WHEN OTHERS THEN  -- handles all other errors
      ROLLBACK;
END;  -- exception handlers and block end here
```

The last example illustrates exception handling, not the effective use of `INSERT` statements. For example, a better way to do the insert follows:

```
INSERT INTO stats (symbol, ratio)
   SELECT symbol, DECODE(earnings, 0, NULL, price / earnings)
   FROM stocks WHERE symbol = 'XYZ';
```

In this example, a subquery supplies values to the `INSERT` statement. If earnings are zero, the function `DECODE` returns a null. Otherwise, `DECODE` returns the price-to-earnings ratio.

Advantages of PL/SQL Exceptions

Using exceptions for error handling has several advantages. Without exception handling, every time you issue a command, you must check for execution errors:

```
BEGIN
   SELECT ...
      -- check for 'no data found' error
   SELECT ...
      -- check for 'no data found' error
   SELECT ...
      -- check for 'no data found' error
```

Error processing is not clearly separated from normal processing; nor is it robust. If you neglect to code a check, the error goes undetected and is likely to cause other, seemingly unrelated errors.

With exceptions, you can handle errors conveniently without the need to code multiple checks, as follows:

```
BEGIN
   SELECT ...
   SELECT ...
   SELECT ...
   ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors
```

Exceptions improve readability by letting you isolate error-handling routines. The primary algorithm is not obscured by error recovery algorithms. Exceptions also improve reliability. You need not worry about checking for an error at every point it might occur. Just add an exception handler to your PL/SQL block. If the exception is ever raised in that block (or any sub-block), you can be sure it will be handled.

Predefined PL/SQL Exceptions

An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception `NO_DATA_FOUND` if a `SELECT INTO` statement returns no rows.

To handle other Oracle errors, you can use the `OTHERS` handler. The functions `SQLCODE` and `SQLERRM` are especially useful in the `OTHERS` handler because they return the Oracle error code and message text. Alternatively, you can use the pragma `EXCEPTION_INIT` to associate exception names with Oracle error codes.

PL/SQL declares predefined exceptions globally in package `STANDARD`, which defines the PL/SQL environment. So, you need not declare them yourself. You can write handlers for predefined exceptions using the names in the following list:

Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51

TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Brief descriptions of the predefined exceptions follow:

Exception	Raised when ...
ACCESS_INTO_NULL	Your program attempts to assign values to the attributes of an uninitialized (atomically null) object.
CASE_NOT_FOUND	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	Your program attempts to apply collection methods other than EXISTS to an uninitialized (atomically null) nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	Your program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers. So, your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	Your program attempts to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	Your program attempts an illegal cursor operation such as closing an unopened cursor.
INVALID_NUMBER	In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE_ERROR is raised.) This exception is also raised when the LIMIT-clause expression in a bulk FETCH statement does not evaluate to a positive number.
LOGIN_DENIED	Your program attempts to log on to Oracle with an invalid username and/or password.
NO_DATA_FOUND	A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table. SQL aggregate functions such as AVG and SUM always return a value or a null. So, a SELECT INTO statement that calls an aggregate function never raises NO_DATA_FOUND. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised.
NOT_LOGGED_ON	Your program issues a database call without being connected to Oracle.
PROGRAM_ERROR	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. For example, when an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.
SELF_IS_NULL	Your program attempts to call a MEMBER method on a null instance. That is, the built-in parameter SELF (which is always the

	first parameter passed to a MEMBER method) is null.
STORAGE_ERROR	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	Your program references a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	Your program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.
SYS_INVALID_ROWID	The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.
TIMEOUT_ON_RESOURCE	A time-out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	A SELECT INTO statement returns more than one row.
VALUE_ERROR	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)
ZERO_DIVIDE	Your program attempts to divide a number by zero.

Defining Your Own PL/SQL Exceptions

PL/SQL lets you define exceptions of your own. Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by `RAISE` statements.

Declaring PL/SQL Exceptions

Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword `EXCEPTION`. In the following example, you declare an exception named `past_due`:

```
DECLARE
   past_due EXCEPTION;
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

Scope Rules for PL/SQL Exceptions

You cannot declare an exception twice in the same block. You can, however, 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.

If you redeclare a global exception in a sub-block, the local declaration prevails. So, the sub-block cannot reference the global exception unless it was declared in a labeled block, in which case the following syntax is valid:

```
block_label.exception_name
```

The following example illustrates the scope rules:

```
DECLARE
   past_due EXCEPTION;
   acct_num NUMBER;
BEGIN
   DECLARE  ---------- sub-block begins
      past_due EXCEPTION;  -- this declaration prevails
      acct_num NUMBER;
   BEGIN
      ...
      IF ... THEN
         RAISE past_due;  -- this is not handled
      END IF;
   END;  ------------- sub-block ends
EXCEPTION
   WHEN past_due THEN  -- does not handle RAISEd exception
      ...
END;
```

The enclosing block does not handle the raised exception because the declaration of `past_due` in the sub-block prevails. Though they share the same name, the two `past_due` exceptions are different, just as the two `acct_num` variables share the same name but are different variables. Therefore, the `RAISE` statement and the `WHEN` clause refer to different exceptions. To have the enclosing block handle the raised exception, you must remove its declaration from the sub-block or define an `OTHERS` handler.

Associating a PL/SQL Exception with a Number: Pragma EXCEPTION_INIT

To handle error conditions (typically ORA- messages) that have no predefined name, you must use the OTHERS handler or the pragma EXCEPTION_INIT. A **pragma** is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an **error stack**, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram, or package using the syntax

```
PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);
```

where exception_name is the name of a previously declared exception and the number is a negative value corresponding to an ORA- error number. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in the following example:

```
DECLARE
   deadlock_detected EXCEPTION;
   PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
   ... -- Some operation that causes an ORA-00060 error
EXCEPTION
   WHEN deadlock_detected THEN
      -- handle the error
END;
```

Defining Your Own Error Messages: Procedure RAISE_APPLICATION_ERROR

The procedure RAISE_APPLICATION_ERROR lets you issue user-defined ORA- error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To call RAISE_APPLICATION_ERROR, use the syntax

```
raise_application_error(error_number, message[, {TRUE | FALSE}]);
```

where error_number is a negative integer in the range -20000 .. -20999 and message is a character string up to 2048 bytes long. If the optional third parameter is TRUE, the error is placed on the stack of previous errors. If the parameter is FALSE (the default), the error replaces all previous errors. RAISE_APPLICATION_ERROR is part of package DBMS_STANDARD, and as with package STANDARD, you do not need to qualify references to it.

An application can call raise_application_error only from an executing stored subprogram (or method). When called, raise_application_error ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error.

In the following example, you call raise_application_error if an employee's salary is missing:

```
CREATE PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) AS
   curr_sal NUMBER;
BEGIN
   SELECT sal INTO curr_sal FROM emp WHERE empno = emp_id;
   IF curr_sal IS NULL THEN
      /* Issue user-defined error message. */
      raise_application_error(-20101, 'Salary is missing');
   ELSE
      UPDATE emp SET sal = curr_sal + amount WHERE empno = emp_id;
   END IF;
END raise_salary;
```

The calling application gets a PL/SQL exception, which it can process using the error-reporting functions SQLCODE and SQLERRM in an OTHERS handler. Also, it can use the pragma EXCEPTION_INIT to map specific error numbers returned by raise_application_error to exceptions of its own, as the following Pro*C example shows:

```
EXEC SQL EXECUTE
   /* Execute embedded PL/SQL block using host
      variables my_emp_id and my_amount, which were
      assigned values in the host environment. */
   DECLARE
      null_salary EXCEPTION;
      /* Map error number returned by raise_application_error
         to user-defined exception. */
      PRAGMA EXCEPTION_INIT(null_salary, -20101);
   BEGIN
      raise_salary(:my_emp_id, :my_amount);
   EXCEPTION
      WHEN null_salary THEN
```

```
          INSERT INTO emp_audit VALUES (:my_emp_id, ...);
   END;
END-EXEC;
```

This technique allows the calling application to handle error conditions in specific exception handlers.

Redeclaring Predefined Exceptions

Remember, PL/SQL declares predefined exceptions globally in package `STANDARD`, so you need not declare them yourself. Redeclaring predefined exceptions is error prone because your local declaration overrides the global declaration. For example, if you declare an exception named *invalid_number* and then PL/SQL raises the predefined exception `INVALID_NUMBER` internally, a handler written for `INVALID_NUMBER` will not catch the internal exception. In such cases, you must use dot notation to specify the predefined exception, as follows:

```
EXCEPTION
   WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN
      -- handle the error
END;
```

How PL/SQL Exceptions Are Raised

Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that you have associated with an Oracle error number using `EXCEPTION_INIT`. However, other user-defined exceptions must be raised explicitly by `RAISE` statements.

Raising Exceptions with the RAISE Statement

PL/SQL blocks and subprograms should raise an exception only when an error makes it undesirable or impossible to finish processing. You can place `RAISE` statements for a given exception anywhere within the scope of that exception. In the following example, you alert your PL/SQL block to a user-defined exception named `out_of_stock`:

```
DECLARE
   out_of_stock   EXCEPTION;
   number_on_hand NUMBER(4);
BEGIN
   ...
   IF number_on_hand < 1 THEN
```

```
      RAISE out_of_stock;
   END IF;
EXCEPTION
   WHEN out_of_stock THEN
      -- handle the error
END;
```

You can also raise a predefined exception explicitly. That way, an exception handler written for the predefined exception can process other errors, as the following example shows:

```
DECLARE
   acct_type INTEGER := 7;
BEGIN
   IF acct_type NOT IN (1, 2, 3) THEN
      RAISE INVALID_NUMBER;  -- raise predefined exception
   END IF;
EXCEPTION
   WHEN INVALID_NUMBER THEN
      ROLLBACK;
END;
```

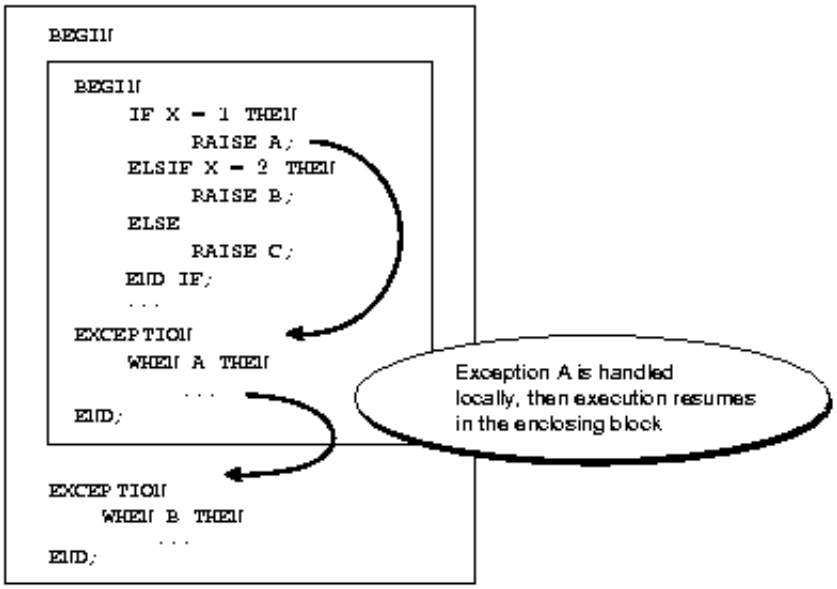
How PL/SQL Exceptions Propagate

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 reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search. In the latter case, PL/SQL returns an *unhandled exception* error to the host environment.

However, exceptions cannot propagate across remote procedure calls (RPCs). Therefore, a PL/SQL block cannot catch an exception raised by a remote subprogram. For a workaround, see "Defining Your Own Error Messages: Procedure RAISE_APPLICATION_ERROR".

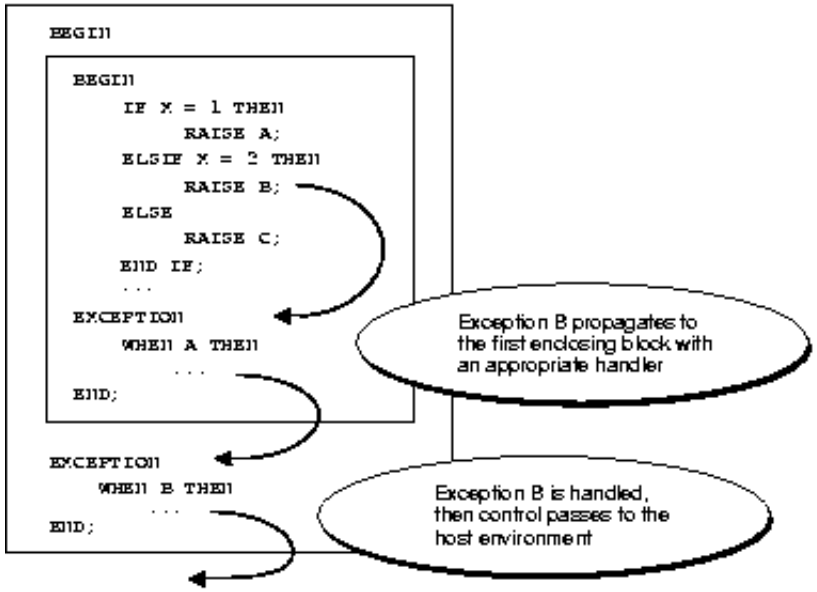
Figure 7-1, Figure 7-2, and Figure 7-3 illustrate the basic propagation rules.

Figure 7-1 Propagation Rules: Example 1



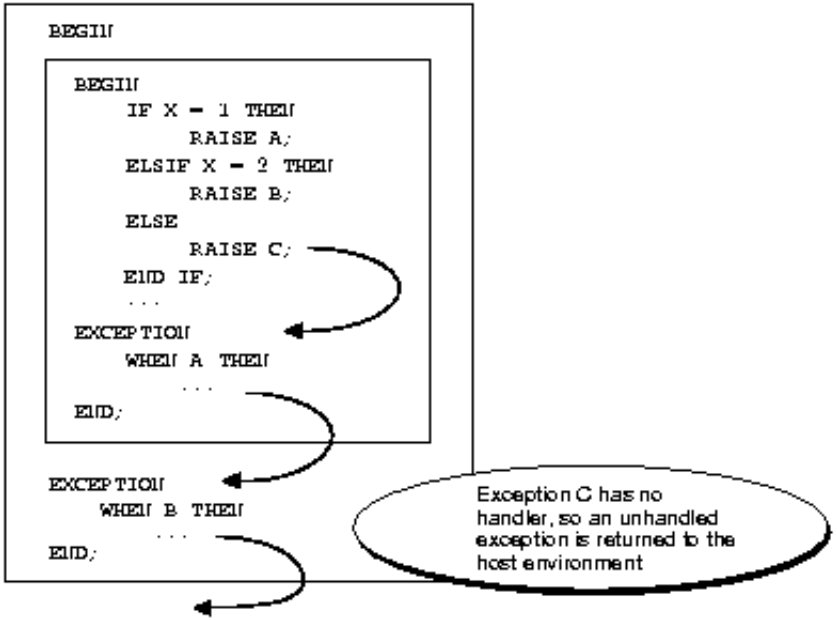
Text description of the illustration pls81009_propagation_rules_example1.gif

Figure 7-2 Propagation Rules: Example 2



Text description of the illustration pls81010_propagation_rules_example2.gif

Figure 7-3 Propagation Rules: Example 3



Text description of the illustration pls81011_propagation_rules_example3.gif

An exception can propagate beyond its scope, that is, beyond the block in which it was declared. Consider the following example:

```
BEGIN
   ...
   DECLARE  ---------- sub-block begins
      past_due EXCEPTION;
   BEGIN
      ...
      IF ... THEN
         RAISE past_due;
      END IF;
   END;  ------------- sub-block ends
EXCEPTION
   ...
   WHEN OTHERS THEN
```

```
      ROLLBACK;
END;
```

Because the block in which exception `past_due` was declared has no handler for it, the exception propagates to the enclosing block. But, according to the scope rules, enclosing blocks cannot reference exceptions declared in a sub-block. So, only an `OTHERS` handler can catch the exception. If there is no handler for a user-defined exception, the calling application gets the following error:

```
ORA-06510: PL/SQL: unhandled user-defined exception
```

Reraising a PL/SQL Exception

Sometimes, you want to *reraise* an exception, that is, handle it locally, then pass it to an enclosing block. For example, you might want to roll back a transaction in the current block, then log the error in an enclosing block.

To reraise an exception, simply place a `RAISE` statement in the local handler, as shown in the following example:

```
DECLARE
   out_of_balance  EXCEPTION;
BEGIN
   ...
   BEGIN  ---------- sub-block begins
      ...
      IF ... THEN
         RAISE out_of_balance;  -- raise the exception
      END IF;
   EXCEPTION
      WHEN out_of_balance THEN
         -- handle the error
         RAISE;  -- reraise the current exception
   END;  ------------ sub-block ends
EXCEPTION
   WHEN out_of_balance THEN
      -- handle the error differently
   ...
END;
```

Omitting the exception name in a RAISE statement--allowed only in an exception handler--reraises the current exception.

Handling Raised PL/SQL Exceptions

When an exception is raised, normal execution of your PL/SQL block or subprogram stops and control transfers to its exception-handling part, which is formatted as follows:

```
EXCEPTION
   WHEN exception_name1 THEN  -- handler
      sequence_of_statements1
   WHEN exception_name2 THEN  -- another handler
      sequence_of_statements2
   ...
   WHEN OTHERS THEN           -- optional handler
      sequence_of_statements3
END;
```

To catch raised exceptions, you write exception handlers. Each handler consists of a WHEN clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised. These statements complete execution of the block or subprogram; control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions not named specifically. Thus, a block or subprogram can have only one OTHERS handler.

As the following example shows, use of the OTHERS handler guarantees that *no* exception will go unhandled:

```
EXCEPTION
   WHEN ... THEN
      -- handle the error
   WHEN ... THEN
      -- handle the error
   WHEN OTHERS THEN
      -- handle all other errors
END;
```

If you want two or more exceptions to execute the same sequence of statements, list the exception names in the WHEN clause, separating them by the keyword

OR, as follows:

```
EXCEPTION
   WHEN over_limit OR under_limit OR VALUE_ERROR THEN
      -- handle the error
```

If any of the exceptions in the list is raised, the associated sequence of statements is executed. The keyword OTHERS cannot appear in the list of exception names; it must appear by itself. You can have any number of exception handlers, and each handler can associate a list of exceptions with a sequence of statements. However, an exception name can appear only once in the exception-handling part of a PL/SQL block or subprogram.

The usual scoping rules for PL/SQL variables apply, so you can reference local and global variables in an exception handler. However, when an exception is raised inside a cursor FOR loop, the cursor is closed implicitly before the handler is invoked. Therefore, the values of explicit cursor attributes are *not* available in the handler.

Handling Exceptions Raised in Declarations

Exceptions can be raised in declarations by faulty initialization expressions. For example, the following declaration raises an exception because the constant credit_limit cannot store numbers larger than 999:

```
DECLARE
   credit_limit CONSTANT NUMBER(3) := 5000;  -- raises an exception
BEGIN
   ...
EXCEPTION
   WHEN OTHERS THEN -- cannot catch the exception
      ...
END;
```

Handlers in the current block cannot catch the raised exception because an exception raised in a declaration propagates *immediately* to the enclosing block.

Handling Exceptions Raised in Handlers

Only one exception at a time can be active in the exception-handling part of a block or subprogram. So, an exception raised inside a handler propagates immediately to the enclosing block, which is searched to find a handler for the newly raised exception. From there on, the exception propagates normally. Consider the following example:

```
EXCEPTION
   WHEN INVALID_NUMBER THEN
      INSERT INTO ...  -- might raise DUP_VAL_ON_INDEX
   WHEN DUP_VAL_ON_INDEX THEN ...  -- cannot catch the exception
END;
```

Branching to or from an Exception Handler

A `GOTO` statement cannot branch into an exception handler. Also, a `GOTO` statement cannot branch from an exception handler into the current block. For example, the following `GOTO` statement is illegal:

```
DECLARE
   pe_ratio NUMBER(3,1);
BEGIN
   DELETE FROM stats WHERE symbol = 'XYZ';
   SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ';
   <<my_label>>
   INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
   WHEN ZERO_DIVIDE THEN
      pe_ratio := 0;
      GOTO my_label;  -- illegal branch into current block
END;
```

However, a `GOTO` statement can branch from an exception handler into an enclosing block.

Retrieving the Error Code and Error Message: SQLCODE and SQLERRM

In an exception handler, you can use the built-in functions `SQLCODE` and `SQLERRM` to find out which error occurred and to get the associated error message. For internal exceptions, `SQLCODE` returns the number of the Oracle error. The number that `SQLCODE` returns is negative unless the Oracle error is *no data found*, in which case `SQLCODE` returns +100. `SQLERRM` returns the corresponding error message. The message begins with the Oracle error code.

For user-defined exceptions, `SQLCODE` returns +1 and `SQLERRM` returns the message: `User-Defined Exception`.

unless you used the pragma `EXCEPTION_INIT` to associate the exception name with an Oracle error number, in which case `SQLCODE` returns that error number and `SQLERRM` returns the corresponding error message. The maximum length of an Oracle error message is 512 characters including the error code, nested

messages, and message inserts such as table and column names.

If no exception has been raised, SQLCODE returns zero and SQLERRM returns the message: ORA-0000: normal, successful completion.

You can pass an error number to SQLERRM, in which case SQLERRM returns the message associated with that error number. Make sure you pass negative error numbers to SQLERRM. In the following example, you pass positive numbers and so get unwanted results:

```
DECLARE
   err_msg VARCHAR2(100);
BEGIN
   /* Get all Oracle error messages. */
   FOR err_num IN 1..9999 LOOP
      err_msg := SQLERRM(err_num); -- wrong; should be -err_num
      INSERT INTO errors VALUES (err_msg);
   END LOOP;
END;
```

Passing a positive number to SQLERRM always returns the message *user-defined exception* unless you pass +100, in which case SQLERRM returns the message *no data found*. Passing a zero to SQLERRM always returns the message *normal, successful completion*.

You cannot use SQLCODE or SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
   err_num NUMBER;
   err_msg VARCHAR2(100);
BEGIN
   ...
EXCEPTION
   WHEN OTHERS THEN
      err_num := SQLCODE;
      err_msg := SUBSTR(SQLERRM, 1, 100);
      INSERT INTO errors VALUES (err_num, err_msg);
END;
```

The string function SUBSTR ensures that a VALUE_ERROR exception (for truncation) is not raised when you assign the value of SQLERRM to err_msg. The functions SQLCODE and SQLERRM are especially useful in the OTHERS exception handler because they tell you which internal exception was raised.

Note: When using pragma `RESTRICT_REFERENCES` to assert the purity of a stored function, you cannot specify the constraints `WNPS` and `RNPS` if the function calls `SQLCODE` or `SQLERRM`.

Catching Unhandled Exceptions

Remember, if it cannot find a handler for a raised exception, PL/SQL returns an unhandled exception error to the host environment, which determines the outcome. For example, in the Oracle Precompilers environment, any database changes made by a failed SQL statement or PL/SQL block are rolled back.

Unhandled exceptions can also affect subprograms. If you exit a subprogram successfully, PL/SQL assigns values to `OUT` parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to `OUT` parameters (unless they are `NOCOPY` parameters). Also, if a stored subprogram fails with an unhandled exception, PL/SQL does *not* roll back database work done by the subprogram.

You can avoid unhandled exceptions by coding an `OTHERS` handler at the topmost level of every PL/SQL program.

Tips for Handling PL/SQL Errors

In this section, you learn three techniques that increase flexibility.

Continuing after an Exception Is Raised

An exception handler lets you recover from an otherwise fatal error before exiting a block. But when the handler completes, the block is terminated. You cannot return to the current block from an exception handler. In the following example, if the `SELECT INTO` statement raises `ZERO_DIVIDE`, you cannot resume with the `INSERT` statement:

```
DECLARE
   pe_ratio NUMBER(3,1);
BEGIN
   DELETE FROM stats WHERE symbol = 'XYZ';
   SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ';
   INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
   WHEN ZERO_DIVIDE THEN
      ...
END;
```

You can still handle an exception for a statement, then continue with the next statement. Place the statement in its own sub-block with its own exception handlers. If an error occurs in the sub-block, a local handler can catch the exception. When the sub-block ends, the enclosing block continues to execute at the point where the sub-block ends. Consider the following example:

```
DECLARE
   pe_ratio NUMBER(3,1);
BEGIN
   DELETE FROM stats WHERE symbol = 'XYZ';
   BEGIN  ---------- sub-block begins
      SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
         WHERE symbol = 'XYZ';
   EXCEPTION
      WHEN ZERO_DIVIDE THEN
         pe_ratio := 0;
   END;  ---------- sub-block ends
   INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
   WHEN OTHERS THEN
      ...
END;
```

In this example, if the `SELECT INTO` statement raises a `ZERO_DIVIDE` exception, the local handler catches it and sets `pe_ratio` to zero. Execution of the handler is complete, so the sub-block terminates, and execution continues with the `INSERT` statement.

You can also perform a sequence of DML operations where some might fail, and process the exceptions only after the entire operation is complete, as described in "Handling FORALL Exceptions with the %BULK_EXCEPTIONS Attribute".

Retrying a Transaction

After an exception is raised, rather than abandon your transaction, you might want to retry it. The technique is:

1. Encase the transaction in a sub-block.
2. Place the sub-block inside a loop that repeats the transaction.
3. Before starting the transaction, mark a savepoint. If the transaction succeeds, commit, then exit from the loop. If the transaction fails, control transfers to the exception handler, where you roll back to the savepoint undoing any changes, then try to fix the problem.

Consider the example below. When the exception handler completes, the sub-block terminates, control transfers to the `LOOP` statement in the enclosing block, the sub-block starts executing again, and the transaction is retried. You might want to use a `FOR` or `WHILE` loop to limit the number of tries.

```
DECLARE
   name   VARCHAR2(20);
   ans1   VARCHAR2(3);
   ans2   VARCHAR2(3);
   ans3   VARCHAR2(3);
   suffix NUMBER := 1;
BEGIN
   ...
   LOOP  -- could be FOR i IN 1..10 LOOP to allow ten tries
      BEGIN  -- sub-block begins
         SAVEPOINT start_transaction;  -- mark a savepoint
         /* Remove rows from a table of survey results. */
         DELETE FROM results WHERE answer1 = 'NO';
         /* Add a survey respondent's name and answers. */
         INSERT INTO results VALUES (name, ans1, ans2, ans3);
 -- raises DUP_VAL_ON_INDEX if two respondents have the same name
         COMMIT;
         EXIT;
      EXCEPTION
         WHEN DUP_VAL_ON_INDEX THEN
            ROLLBACK TO start_transaction;  -- undo changes
            suffix := suffix + 1;           -- try to fix problem
            name := name || TO_CHAR(suffix);
      END;  -- sub-block ends
   END LOOP;
END;
```

Using Locator Variables to Identify Exception Locations

Using one exception handler for a sequence of statements can mask the statement that caused an error:

```
BEGIN
   SELECT ...
   SELECT ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN ...
      -- Which SELECT statement caused the error?
```

```
END;
```

Normally, this is not a problem. But, if the need arises, you can use a *locator variable* to track statement execution, as follows:

```
DECLARE
   stmt INTEGER := 1;  -- designates 1st SELECT statement
BEGIN
   SELECT ...
   stmt := 2;  -- designates 2nd SELECT statement
   SELECT ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO errors VALUES ('Error in statement ' || stmt);
END;
```

Previous Next

Home Book List Contents Index Master Index Feedback