**10** **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.

**Overview of PL/SQL Runtime Error Handling**

In PL/SQL, an error condition is called an *exception*. Exceptions can be internally defined (by the runtime 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.

The following example calculates a price-to-earnings ratio for a company. If the company has zero earnings, the division operation raises the predefined exception ZERO\_DIVIDE, the execution of the block is interrupted, and control is transferred to the exception handlers. The optional OTHERS handler catches all exceptions that the block does not name specifically.

SET SERVEROUTPUT ON;

DECLARE

stock\_price NUMBER := 9.73;

net\_earnings NUMBER := 0;

pe\_ratio NUMBER;

BEGIN

-- Calculation might cause division-by-zero error.

pe\_ratio := stock\_price / net\_earnings;

dbms\_output.put\_line('Price/earnings ratio = ' || pe\_ratio);

EXCEPTION -- exception handlers begin

-- Only one of the WHEN blocks is executed.

WHEN ZERO\_DIVIDE THEN -- handles 'division by zero' error

dbms\_output.put\_line('Company must have had zero earnings.');

pe\_ratio := null;

WHEN OTHERS THEN -- handles all other errors

dbms\_output.put\_line('Some other kind of error occurred.');

pe\_ratio := null;

END; -- exception handlers and block end here

/

The last example illustrates exception handling. With some better error checking, we could have avoided the exception entirely, by substituting a null for the answer if the denominator was zero:

DECLARE

stock\_price NUMBER := 9.73;

net\_earnings NUMBER := 0;

pe\_ratio NUMBER;

BEGIN

pe\_ratio :=

case net\_earnings

when 0 then null

else stock\_price / net\_earnings

end;

END;

/

**Guidelines for Avoiding and Handling PL/SQL Errors and Exceptions**

Because reliability is crucial for database programs, use both error checking and exception handling to ensure your program can handle all possibilities:

* Add exception handlers whenever there is any possibility of an error occurring. Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors could also occur at other times, for example if a hardware failure with disk storage or memory causes a problem that has nothing to do with your code; but your code still needs to take corrective action.
* Add error-checking code whenever you can predict that an error might occur if your code gets bad input data. Expect that at some time, your code will be passed incorrect or null parameters, that your queries will return no rows or more rows than you expect.
* Make your programs robust enough to work even if the database is not in the state you expect. For example, perhaps a table you query will have columns added or deleted, or their types changed. You can avoid such problems by declaring individual variables with %TYPE qualifiers, and declaring records to hold query results with %ROWTYPE qualifiers.
* Handle named exceptions whenever possible, instead of using WHEN OTHERS in exception handlers. Learn the names and causes of the predefined exceptions. If your database operations might cause particular ORA- errors, associate names with these errors so you can write handlers for them. (You will learn how to do that later in this chapter.)
* Test your code with different combinations of bad data to see what potential errors arise.
* Write out debugging information in your exception handlers. You might store such information in a separate table. If so, do it by making a call to a procedure declared with the PRAGMA AUTONOMOUS\_TRANSACTION, so that you can commit your debugging information, even if you roll back the work that the main procedure was doing.
* Carefully consider whether each exception handler should commit the transaction, roll it back, or let it continue. Remember, no matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data.

**Advantages of PL/SQL Exceptions**

Using exceptions for error handling has several advantages.

With exceptions, you can reliably handle potential errors from many statements with a single exception handler:

BEGIN

SELECT ...

SELECT ...

procedure\_that\_performs\_select();

...

EXCEPTION

WHEN NO\_DATA\_FOUND THEN -- catches all 'no data found' errors

Instead of 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.

Sometimes the error is not immediately obvious, and could not be detected until later when you perform calculations using bad data. Again, a single exception handler can trap all division-by-zero errors, bad array subscripts, and so on.

If you need to check for errors at a specific spot, you can enclose a single statement or a group of statements inside its own BEGIN-END block with its own exception handler. You can make the checking as general or as precise as you like.

Isolating error-handling routines makes the rest of the program easier to read and understand.

**Summary of Predefined PL/SQL Exceptions**

An internal exception is raised automatically if your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. 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.

You can use the pragma EXCEPTION\_INIT to associate exception names with other Oracle error codes that you can anticipate. To handle unexpected Oracle errors, you can use the OTHERS handler. Within this handler, you can call the functions SQLCODE and SQLERRM to return the Oracle error code and message text. Once you know the error code, you can use it with pragma EXCEPTION\_INIT and write a handler specifically for that error.

PL/SQL declares predefined exceptions globally in package STANDARD. 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 | A program attempts to assign values to the attributes of an uninitialized 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 | A program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| CURSOR\_ALREADY\_OPEN | A 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 | A program attempts to store duplicate values in a database column that is constrained by a unique index. |
| INVALID\_CURSOR | A program attempts a cursor operation that is not allowed, 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 | A program attempts to log on to Oracle with an invalid username 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.  Because this exception is used internally by some SQL functions to signal that they are finished, you should not rely on this exception being propagated if you raise it within a function that is called as part of a query. |
| NOT\_LOGGED\_ON | A 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 | A program attempts to call a MEMBER method, but the instance of the object type has not been initialized. The built-in parameter SELF points to the object, and is always the first parameter passed to a MEMBER method. |
| STORAGE\_ERROR | PL/SQL runs out of memory or memory has been corrupted. |
| SUBSCRIPT\_BEYOND\_COUNT | A program references a nested table or varray element using an index number larger than the number of elements in the collection. |
| SUBSCRIPT\_OUTSIDE\_LIMIT | A 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 | A 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. The sub-block cannot reference the global exception, unless the exception is declared in a labeled block and you qualify its name with the block label:

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;

due\_date DATE := SYSDATE - 1;

todays\_date DATE := SYSDATE;

BEGIN

IF due\_date < todays\_date THEN

RAISE past\_due; -- this is not handled

END IF;

END; ------------- sub-block ends

EXCEPTION

WHEN past\_due THEN -- does not handle RAISEd exception

dbms\_output.put\_line('Handling PAST\_DUE exception.');

WHEN OTHERS THEN

dbms\_output.put\_line('Could not recognize PAST\_DUE\_EXCEPTION in this scope.');

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. Thus, 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

null; -- Some operation that causes an ORA-00060 error

EXCEPTION

WHEN deadlock\_detected THEN

null; -- 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 error condition of your choosing happens (in this case, if the current schema owns less than 1000 tables):

DECLARE

num\_tables NUMBER;

BEGIN

SELECT COUNT(\*) INTO num\_tables FROM USER\_TABLES;

IF num\_tables < 1000 THEN

/\* Issue your own error code (ORA-20101) with your own error message. \*/

raise\_application\_error(-20101, 'Expecting at least 1000 tables');

ELSE

NULL; -- Do the rest of the processing (for the non-error case).

END IF;

END;

/

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 := 0;

BEGIN

IF number\_on\_hand < 1 THEN

RAISE out\_of\_stock; -- raise an exception that we defined

END IF;

EXCEPTION

WHEN out\_of\_stock THEN

-- handle the error

dbms\_output.put\_line('Encountered out-of-stock 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

dbms\_output.put\_line('Handling invalid input by rolling back.');

ROLLBACK;

END;

/

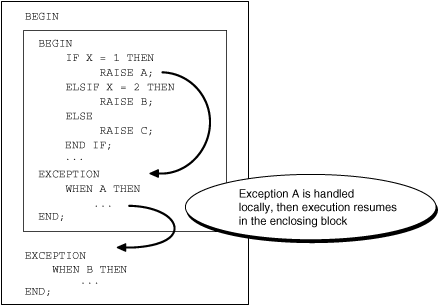
**How PL/SQL Exceptions Propagate**

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. If no handler is found, PL/SQL returns an *unhandled exception* error to the host environment.

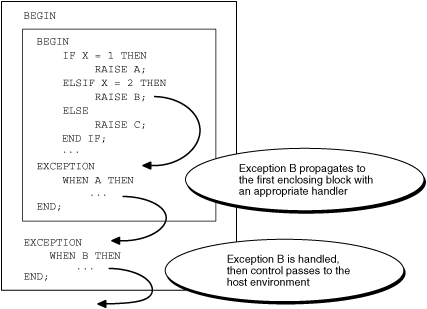
Exceptions cannot propagate across remote procedure calls done through database links. A PL/SQL block cannot catch an exception raised by a remote subprogram. For a workaround, see ["Defining Your Own Error Messages: Procedure RAISE\_APPLICATION\_ERROR"](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/07_errs.htm#i1871).

[Figure 10-1](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/07_errs.htm#i5473), [Figure 10-2](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/07_errs.htm#i5479), and [Figure 10-3](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/07_errs.htm#i2641) illustrate the basic propagation rules.

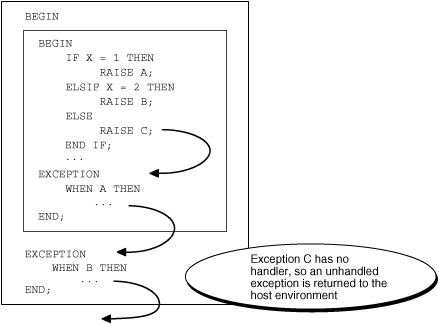
***Figure 10-1 Propagation Rules: Example 1***

  
[Description of the illustration lnpls009.gif](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/img_text/lnpls009.htm)

***Figure 10-2 Propagation Rules: Example 2***

  
[Description of the illustration lnpls010.gif](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/img_text/lnpls010.htm)

***Figure 10-3 Propagation Rules: Example 3***

  
[Description of the illustration lnpls011.gif](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/img_text/lnpls011.htm)

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;

due\_date DATE := trunc(SYSDATE) - 1;

todays\_date DATE := trunc(SYSDATE);

BEGIN

IF due\_date < todays\_date THEN

RAISE past\_due;

END IF;

END; ------------- sub-block ends

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END;

/

Because the block that declares the exception past\_due has no handler for it, the exception propagates to the enclosing block. But the enclosing block cannot reference the name PAST\_DUE, because the scope where it was declared no longer exists. Once the exception name is lost, only an OTHERS handler can catch the exception. If there is no handler for a user-defined exception, the calling application gets this 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, use a RAISE statement without an exception name, which is allowed only in an exception handler:

DECLARE

salary\_too\_high EXCEPTION;

current\_salary NUMBER := 20000;

max\_salary NUMBER := 10000;

erroneous\_salary NUMBER;

BEGIN

BEGIN ---------- sub-block begins

IF current\_salary > max\_salary THEN

RAISE salary\_too\_high; -- raise the exception

END IF;

EXCEPTION

WHEN salary\_too\_high THEN

-- first step in handling the error

dbms\_output.put\_line('Salary ' || erroneous\_salary ||

' is out of range.');

dbms\_output.put\_line('Maximum salary is ' || max\_salary || '.');

RAISE; -- reraise the current exception

END; ------------ sub-block ends

EXCEPTION

WHEN salary\_too\_high THEN

-- handle the error more thoroughly

erroneous\_salary := current\_salary;

current\_salary := max\_salary;

dbms\_output.put\_line('Revising salary from ' || erroneous\_salary ||

'to ' || current\_salary || '.');

END;

/

**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

NULL;

EXCEPTION

WHEN OTHERS THEN

-- Cannot catch the exception. This handler is never called.

dbms\_output.put\_line('Can''t handle an exception in a declaration.');

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**

When an exception occurs within an exception handler, that same handler cannot catch the exception. An exception raised inside a handler propagates immediately to the enclosing block, which is searched to find a handler for this new exception. From there on, the exception propagates normally. For 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 can branch from an exception handler into an enclosing block.

A GOTO statement cannot branch into an exception handler, or from an exception handler into the current 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.

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\_msg VARCHAR2(100);

BEGIN

/\* Get a few Oracle error messages. \*/

FOR err\_num IN 1..3 LOOP

err\_msg := SUBSTR(SQLERRM(-err\_num),1,100);

dbms\_output.put\_line('Error number = ' || err\_num);

dbms\_output.put\_line('Error message = ' || err\_msg);

END LOOP;

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

NULL;

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

NULL;

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"](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/12_tune.htm#i49099).

**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.

In the following example, the INSERT statement might raise an exception because of a duplicate value in a unique column. In that case, we change the value that needs to be unique and continue with the next loop iteration. If the INSERT succeeds, we exit from the loop immediately. With this technique, you should use a FOR or WHILE loop to limit the number of attempts.

DECLARE

name VARCHAR2(20);

ans1 VARCHAR2(3);

ans2 VARCHAR2(3);

ans3 VARCHAR2(3);

suffix NUMBER := 1;

BEGIN

FOR i IN 1..10 LOOP -- try 10 times

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, such as INSERT, DELETE, or UPDATE statements, can mask the statement that caused an error. If you need to know which statement failed, you can use a *locator variable*:

DECLARE

stmt INTEGER;

name VARCHAR2(100);

BEGIN

stmt := 1; -- designates 1st SELECT statement

SELECT table\_name INTO name FROM user\_tables WHERE table\_name LIKE 'ABC%';

stmt := 2; -- designates 2nd SELECT statement

SELECT table\_name INTO name FROM user\_tables WHERE table\_name LIKE 'XYZ%';

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

dbms\_output.put\_line('Table name not found in query ' || stmt);

END;

/

**Overview of PL/SQL Compile-Time Warnings**

To make your programs more robust and avoid problems at run time, you can turn on checking for certain warning conditions. These conditions are not serious enough to produce an error and keep you from compiling a subprogram. They might point out something in the subprogram that produces an undefined result or might create a performance problem.

To work with PL/SQL warning messages, you use the PLSQL\_WARNINGS initialization parameter, the DBMS\_WARNING package, and the USER/DBA/ALL\_PLSQL\_OBJECT\_SETTINGS views.

**PL/SQL Warning Categories**

PL/SQL warning messages are divided into categories, so that you can suppress or display groups of similar warnings during compilation. The categories are:

**Severe**: Messages for conditions that might cause unexpected behavior or wrong results, such as aliasing problems with parameters.

**Performance**: Messages for conditions that might cause performance problems, such as passing a VARCHAR2 value to a NUMBER column in an INSERT statement.

**Informational**: Messages for conditions that do not have an effect on performance or correctness, but that you might want to change to make the code more maintainable, such as dead code that can never be executed.

The keyword **All** is a shorthand way to refer to all warning messages.

You can also treat particular messages as errors instead of warnings. For example, if you know that the warning message PLW-05003 represents a serious problem in your code, including 'ERROR:05003' in the PLSQL\_WARNINGS setting makes that condition trigger an error message (PLS\_05003) instead of a warning message. An error message causes the compilation to fail.

**Controlling PL/SQL Warning Messages**

To let the database issue warning messages during PL/SQL compilation, you set the initialization parameter PLSQL\_WARNINGS. You can enable and disable entire categories of warnings (ALL, SEVERE, INFORMATIONAL, PERFORMANCE), enable and disable specific message numbers, and make the database treat certain warnings as compilation errors so that those conditions must be corrected.

This parameter can be set at the system level or the session level. You can also set it for a single compilation by including it as part of the ALTER PROCEDURE statement. You might turn on all warnings during development, turn off all warnings when deploying for production, or turn on some warnings when working on a particular subprogram where you are concerned with some aspect, such as unnecessary code or performance.

ALTER SYSTEM SET PLSQL\_WARNINGS='ENABLE:ALL'; -- For debugging during development.

ALTER SESSION SET PLSQL\_WARNINGS='ENABLE:PERFORMANCE'; -- To focus on one aspect.

ALTER PROCEDURE hello COMPILE PLSQL\_WARNINGS='ENABLE:PERFORMANCE'; -- Recompile with extra checking.

ALTER SESSION SET PLSQL\_WARNINGS='DISABLE:ALL'; -- To turn off all warnings.

-- We want to hear about 'severe' warnings, don't want to hear about 'performance'

-- warnings, and want PLW-06002 warnings to produce errors that halt compilation.

ALTER SESSION SET PLSQL\_WARNINGS='ENABLE:SEVERE','DISABLE:PERFORMANCE','ERROR:06002';

Warning messages can be issued during compilation of PL/SQL subprograms; anonymous blocks do not produce any warnings.

The settings for the PLSQL\_WARNINGS parameter are stored along with each compiled subprogram. If you recompile the subprogram with a CREATE OR REPLACE statement, the current settings for that session are used. If you recompile the subprogram with an ALTER ... COMPILE statement, the current session setting might be used, or the original setting that was stored with the subprogram, depending on whether you include the REUSE SETTINGS clause in the statement.

To see any warnings generated during compilation, you use the SQL\*Plus SHOW ERRORS command or query the USER\_ERRORS data dictionary view. PL/SQL warning messages all use the prefix PLW.

**Using the DBMS\_WARNING Package**

If you are writing a development environment that compiles PL/SQL subprograms, you can control PL/SQL warning messages by calling subprograms in the DBMS\_WARNING package. You might also use this package when compiling a complex application, made up of several nested SQL\*Plus scripts, where different warning settings apply to different subprograms. You can save the current state of the PLSQL\_WARNINGS parameter with one call to the package, change the parameter to compile a particular set of subprograms, then restore the original parameter value.

For example, here is a procedure with unnecessary code that could be removed. It could represent a mistake, or it could be intentionally hidden by a debug flag, so you might or might not want a warning message for it.

CREATE OR REPLACE PROCEDURE dead\_code

AS

x number := 10;

BEGIN

if x = 10 then

x := 20;

else

x := 100; -- dead code (never reached)

end if;

END dead\_code;/

-- By default, the preceding procedure compiles with no errors or warnings.

-- Now enable all warning messages, just for this session.

CALL DBMS\_WARNING.SET\_WARNING\_SETTING\_STRING('ENABLE:ALL' ,'SESSION');

-- Check the current warning setting.

select dbms\_warning.get\_warning\_setting\_string() from dual;

-- When we recompile the procedure, we will see a warning about the dead code.

ALTER PROCEDURE dead\_code COMPILE;

**See Also:** [ALTER PROCEDURE](http://download.oracle.com/docs/cd/B13789_01/server.101/b10759/statements_2006.htm#SQLRF00812), DBMS\_WARNING package in the [*PL/SQL Packages and Types Reference*](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10802/toc.htm), PLW- messages in the [*Oracle Database Error Messages*](http://download.oracle.com/docs/cd/B13789_01/server.101/b10744/toc.htm)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | [[Go to previous page](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/09_packs.htm) Previous](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/09_packs.htm) | [[Go to next page](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/12_tune.htm) Next](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/12_tune.htm) | | Oracle [Copyright © 1996, 2003 Oracle Corporation](http://download.oracle.com/docs/cd/B13789_01/dcommon/html/cpyr.htm) All Rights Reserved. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | [[Go to Documentation Home](http://download.oracle.com/docs/cd/B13789_01/index.htm) Home](http://download.oracle.com/docs/cd/B13789_01/index.htm) | [[Go to Book List](http://download.oracle.com/docs/cd/B13789_01/nav/portal_3.htm) Book List](http://download.oracle.com/docs/cd/B13789_01/nav/portal_3.htm) | [[Go to Table of Contents](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/toc.htm) Contents](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/toc.htm) | [[Go to Index](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/index.htm) Index](http://download.oracle.com/docs/cd/B13789_01/appdev.101/b10807/index.htm) | [[Go to Master Index](http://download.oracle.com/docs/cd/B13789_01/mix.101/b12039/toc.htm) Master Index](http://download.oracle.com/docs/cd/B13789_01/mix.101/b12039/toc.htm) | [[Go to Feedback page](http://download.oracle.com/docs/cd/B13789_01/dcommon/html/feedback.htm) Feedback](http://download.oracle.com/docs/cd/B13789_01/dcommon/html/feedback.htm) | |