10 Handling PL/SQL Errors

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 contains these topics:

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. For information on managing errors when using `BULK COLLECT`, see "Handling FORALL Exceptions with the %BULK_EXCEPTIONS Attribute".

Example 10-1 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.

Example 10-1 Runtime Error Handling

```
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, as shown in the following example.

```
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:

Example 10-2 Managing Multiple Errors With a Single Exception Handler

```
DECLARE

   emp_column       VARCHAR2(30) := 'last_name';

   table_name       VARCHAR2(30) := 'emp';

   temp_var         VARCHAR2(30);

BEGIN

  temp_var := emp_column;

  SELECT COLUMN_NAME INTO temp_var FROM USER_TAB_COLS

    WHERE TABLE_NAME = 'EMPLOYEES' AND COLUMN_NAME = UPPER(emp_column);

-- processing here

  temp_var := table_name;

  SELECT OBJECT_NAME INTO temp_var FROM USER_OBJECTS

    WHERE OBJECT_NAME = UPPER(table_name) AND OBJECT_TYPE = 'TABLE';

-- processing here

EXCEPTION

   WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors

     DBMS_OUTPUT.PUT_LINE ('No Data found for SELECT on ' || temp_var);

END;

/
```

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 table:

Exception	ORA Error	SQLCODE	Raise When ...
`ACCESS_INTO_NULL`	`06530`	`-6530`	A program attempts to assign values to the attributes of an
`CASE_NOT_FOUND`	`06592`	`-6592`	None of the choices in the `WHEN` clauses of a `CASE` stateme
`COLLECTION_IS_NULL`	`06531`	`-6531`	A program attempts to apply collection methods other tha assign values to the elements of an uninitialized nested tab
`CURSOR_ALREADY_OPEN`	`06511`	`-6511`	A program attempts to open an already open cursor. A cu opens the cursor to which it refers, so your program canno
`DUP_VAL_ON_INDEX`	`00001`	`-1`	A program attempts to store duplicate values in a column
`INVALID_CURSOR`	`01001`	`-1001`	A program attempts a cursor operation that is not allowed
`INVALID_NUMBER`	`01722`	`-1722`	n a SQL statement, the conversion of a character string int procedural statements, `VALUE_ERROR` is raised.) This excep does not evaluate to a positive number.
`LOGIN_DENIED`	`01017`	`-1017`	A program attempts to log on to Oracle with an invalid us
`NO_DATA_FOUND`	`01403`	`+100`	A `SELECT INTO` statement returns no rows, or your progra index-by table. Because this exception is used internally by some SQL functions t within a function that is called as part of a query.
`NOT_LOGGED_ON`	`01012`	`-1012`	A program issues a database call without being connected
`PROGRAM_ERROR`	`06501`	`-6501`	PL/SQL has an internal problem.

Exception	ORA Error	SQLCODE	Raise When ...
ROWTYPE_MISMATCH	06504	-6504	The host cursor variable and PL/SQL cursor variable invo variable is passed to a stored subprogram, the return types
SELF_IS_NULL	30625	-30625	A program attempts to call a MEMBER method, but the insta to the object, and is always the first parameter passed to a
STORAGE_ERROR	06500	-6500	PL/SQL runs out of memory or memory has been corrupt
SUBSCRIPT_BEYOND_COUNT	06533	-6533	A program references a nested table or varray element usi
SUBSCRIPT_OUTSIDE_LIMIT	06532	-6532	A program references a nested table or varray element usi
SYS_INVALID_ROWID	01410	-1410	The conversion of a character string into a universal rowi
TIMEOUT_ON_RESOURCE	00051	-51	A time out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	01422	-1422	A SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	An arithmetic, conversion, truncation, or size-constraint e character variable, if the value is longer than the declared procedural statements, VALUE_ERROR is raised if the conve statements, INVALID_NUMBER is raised.)
ZERO_DIVIDE	01476	-1476	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`

Example 10-3 illustrates the scope rules:

Example 10-3 Scope of PL/SQL Exceptions

```
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 Example 10-4.

Example 10-4 Using PRAGMA EXCEPTION_INIT

```
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 Example 10-5, 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):

Example 10-5 Raising an Application Error With raise_application_error

```
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.

         Note that you do not need to qualify raise_application_error with

         DBMS_STANDARD */

      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 v_emp_id and v_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(:v_emp_id, :v_amount);
  EXCEPTION
    WHEN null_salary THEN
      INSERT INTO emp_audit VALUES (:v_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 Example 10-6, you alert your PL/SQL block to a user-defined exception named `out_of_stock`.

Example 10-6 Using RAISE to Force a User-Defined Exception

```
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 Example 10-7 shows:

Example 10-7 Using RAISE to Force a Pre-Defined Exception

```
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;

/
```

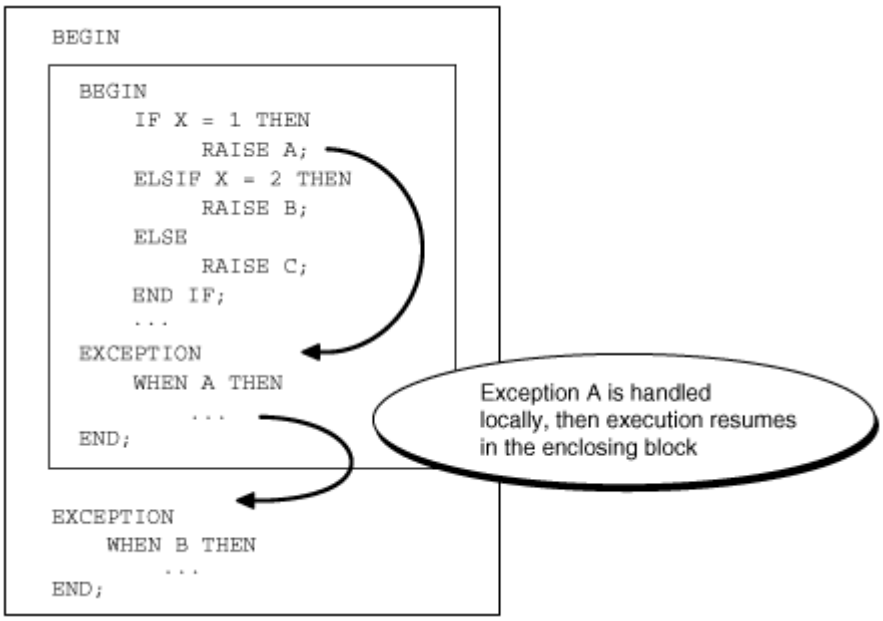
How PL/SQL Exceptions Propagate

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".

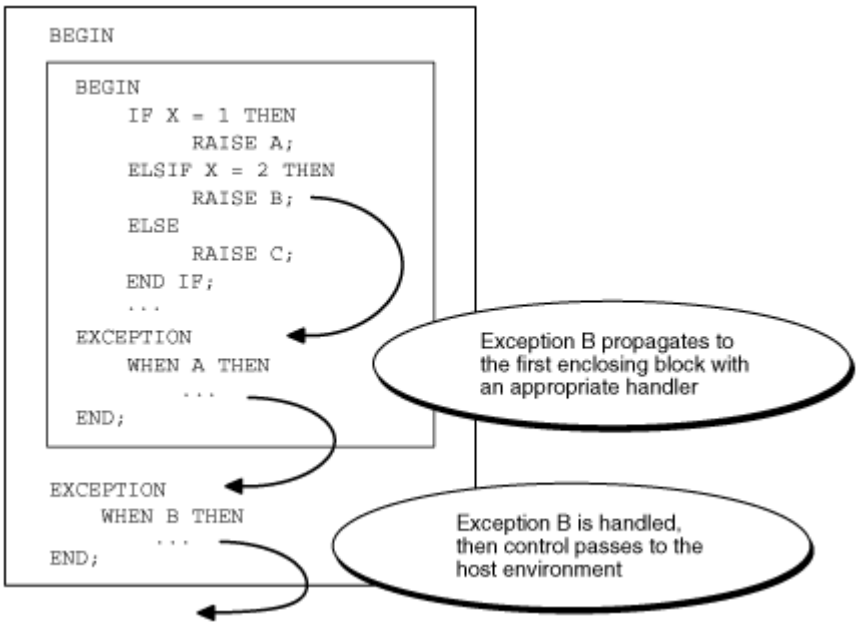
Figure 10-1, Figure 10-2, and Figure 10-3 illustrate the basic propagation rules.

Figure 10-1 Propagation Rules: Example 1



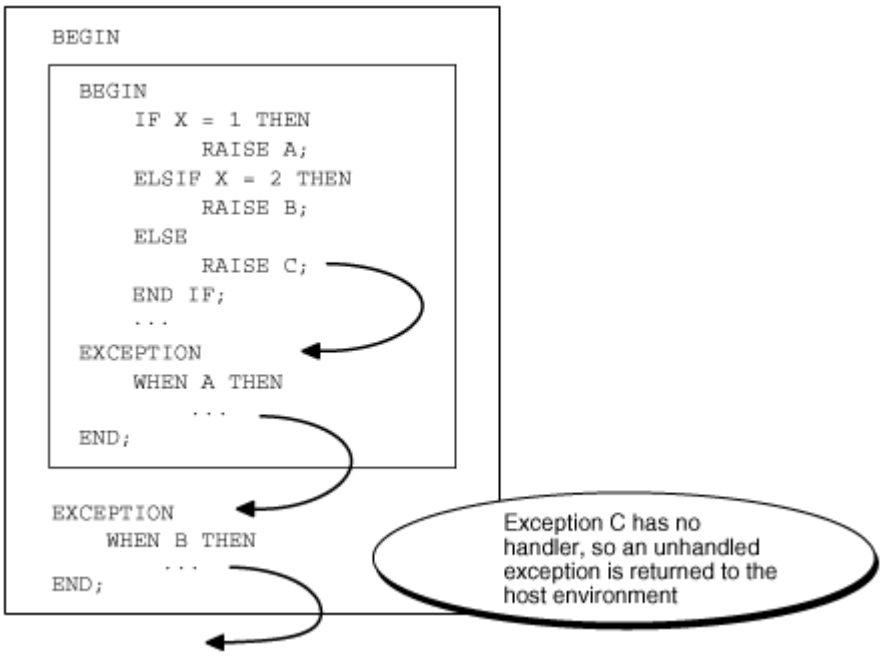
Description of the illustration lnpls009.gif

Figure 10-2 Propagation Rules: Example 2



Description of the illustration lnpls010.gif

Figure 10-3 Propagation Rules: Example 3



Description of the illustration lnpls011.gif

An exception can propagate beyond its scope, that is, beyond the block in which it was declared, as shown in Example 10-8.

Example 10-8 Scope of an Exception

```
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:

Example 10-9 Reraising a PL/SQL Exception

```
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 exception1 THEN -- handler for exception1
    sequence_of_statements1
  WHEN exception2 THEN -- another handler for exception2
    sequence_of_statements2
  ...
  WHEN OTHERS THEN -- optional handler for all other errors
    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. Use of the `OTHERS` handler guarantees that no exception will go unhandled.

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.

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:

Example 10-10 Raising an Exception in a Declaration

```
DECLARE

   credit_limit CONSTANT NUMBER(3) := 5000;  -- raises an error

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 Example 10-11.

Example 10-11 Displaying SQLCODE and SQLERRM

```
CREATE TABLE errors (code NUMBER, message VARCHAR2(64), happened TIMESTAMP);

DECLARE

   name employees.last_name%TYPE;

   v_code NUMBER;

   v_errm VARCHAR2(64);

BEGIN

   SELECT last_name INTO name FROM employees WHERE employee_id = -1;

   EXCEPTION

      WHEN OTHERS THEN

         v_code := SQLCODE;

         v_errm := SUBSTR(SQLERRM, 1 , 64);

         DBMS_OUTPUT.PUT_LINE('Error code ' || v_code || ': ' || v_errm);

-- Normally we would call another procedure, declared with PRAGMA

-- AUTONOMOUS_TRANSACTION, to insert information about errors.

         INSERT INTO errors VALUES (v_code, v_errm, SYSTIMESTAMP);

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.

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 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:

```
CREATE TABLE employees_temp AS

  SELECT employee_id, salary, commission_pct FROM employees;



DECLARE

  sal_calc NUMBER(8,2);

BEGIN

  INSERT INTO employees_temp VALUES (301, 2500, 0);

  SELECT salary / commission_pct INTO sal_calc FROM employees_temp

    WHERE employee_id = 301;

  INSERT INTO employees_temp VALUES (302, sal_calc/100, .1);

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, as shown in Example 10-12.

Example 10-12 Continuing After an Exception

```
DECLARE

  sal_calc NUMBER(8,2);

BEGIN

  INSERT INTO employees_temp VALUES (303, 2500, 0);

  BEGIN -- sub-block begins

    SELECT salary / commission_pct INTO sal_calc FROM employees_temp

      WHERE employee_id = 301;

    EXCEPTION

      WHEN ZERO_DIVIDE THEN

        sal_calc := 2500;

  END; -- sub-block ends

  INSERT INTO employees_temp VALUES (304, sal_calc/100, .1);

EXCEPTION

  WHEN ZERO_DIVIDE THEN

    NULL;

END;

/
```

In this example, if the `SELECT INTO` statement raises a `ZERO_DIVIDE` exception, the local handler catches it and sets `sal_calc` to 2500. Execution of the handler is complete, so the sub-block terminates, and execution continues with the `INSERT` statement. See also Example 5-38, "Collection Exceptions".

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.

In Example 10-13, 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.

Example 10-13 Retrying a Transaction After an Exception

```
CREATE TABLE results ( res_name VARCHAR(20), res_answer VARCHAR2(3) );

CREATE UNIQUE INDEX res_name_ix ON results (res_name);

INSERT INTO results VALUES ('SMYTHE', 'YES');

INSERT INTO results VALUES ('JONES', 'NO');



DECLARE

   name     VARCHAR2(20) := 'SMYTHE';

   answer   VARCHAR2(3) := 'NO';

   suffix   NUMBER := 1;

BEGIN

   FOR i IN 1..5 LOOP  -- try 5 times

      BEGIN  -- sub-block begins

         SAVEPOINT start_transaction;  -- mark a savepoint

         /* Remove rows from a table of survey results. */

         DELETE FROM results WHERE res_answer = 'NO';

         /* Add a survey respondent's name and answers. */

         INSERT INTO results VALUES (name, answer);

 -- 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`:

Example 10-14 Using a Locator Variable to Identify the Location of an Exception

```
CREATE OR REPLACE PROCEDURE loc_var AS
   stmt_no NUMBER;
   name    VARCHAR2(100);
BEGIN
   stmt_no := 1;  -- designates 1st SELECT statement
   SELECT table_name INTO name FROM user_tables WHERE table_name LIKE 'ABC%';
   stmt_no := 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_no);
END;
```

```
/

CALL loc_var();
```

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 unreachable 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` ... `COMPILE` 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.

Example 10-15 Controlling the Display of PL/SQL Warnings

```
-- To focus on one aspect

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';

-- Recompile with extra checking

ALTER PROCEDURE loc_var COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE'

  REUSE SETTINGS;

-- To turn off all warnings

ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';

-- Display 'severe' warnings, don't want 'performance' warnings, and

-- want PLW-06002 warnings to produce errors that halt compilation

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE', 'DISABLE:PERFORMANCE',

                                 'ERROR:06002';

-- For debugging during development

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

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. For more information, see `ALTER FUNCTION`, `ALTER PACKAGE`, and `ALTER PROCEDURE` in *Oracle Database SQL Reference*.

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, Example 10-16 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.

Example 10-16 Using the DBMS_WARNING Package to Display Warnings

```
-- When warnings disabled, the following procedure compiles with no warnings

CREATE OR REPLACE PROCEDURE unreachable_code AS

  x CONSTANT BOOLEAN := TRUE;

BEGIN

  IF x THEN

    DBMS_OUTPUT.PUT_LINE('TRUE');

  ELSE

    DBMS_OUTPUT.PUT_LINE('FALSE');

  END IF;

END unreachable_code;

/

-- enable all warning messages 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;


-- Recompile the procedure and a warning about unreachable code displays

ALTER PROCEDURE unreachable_code COMPILE;

SHOW ERRORS;
```

In Example 10-16, you could have used the following `ALTER PROCEDURE` without the call to `DBMS_WARNINGS.set_warning_setting_string`:

```
ALTER PROCEDURE unreachable_code COMPILE
    PLSQL_WARNINGS = 'ENABLE:ALL' REUSE SETTINGS;
```

For more information, see `ALTER PROCEDURE` in *Oracle Database SQL Reference*, `DBMS_WARNING` package in *Oracle Database PL/SQL Packages and Types Reference*, and PLW- messages in *Oracle Database Error Messages*