CHAPTER 6

Exception Handlers

It is a sad fact of life that many programmers rarely take the time to properly bulletproof their programs. Instead, wishful thinking often reigns. Most of us find it hard enough —and more than enough work—to simply write the code that implements the positive aspects of an application: maintaining customers, generating invoices, and so on. It is devilishly difficult, from both a psychological standpoint and a resources perspective, to focus on the negative: for example, what happens when the user presses the wrong key? If the database is unavailable, what should I do?

As a result, we write applications that assume the best of all possible worlds, hoping that our programs are bug-free, that users will enter the correct data in the correct fashion, and that all systems (hardware and software) will always be a "go."

Of course, harsh reality dictates that no matter how hard you try, there will always be one more bug in your application. And your users will somehow always find just the right sequence of keystrokes to make a form implode. The challenge is clear: either you spend the time up front to properly debug and bulletproof your programs, or you fight an unending series of rear-guard battles, taking frantic calls from your users and putting out the fires.

You know what you should do. Fortunately, PL/SQL offers a powerful and flexible way to trap and handle errors. It is entirely feasible within the PL/SQL language to build an application that fully protects the user and the database from errors.

Exception-Handling Concepts and Terminology

In the PL/SQL language, errors of any kind are treated as *exceptions*—situations that should not occur—in your program. An exception can be one of the following:

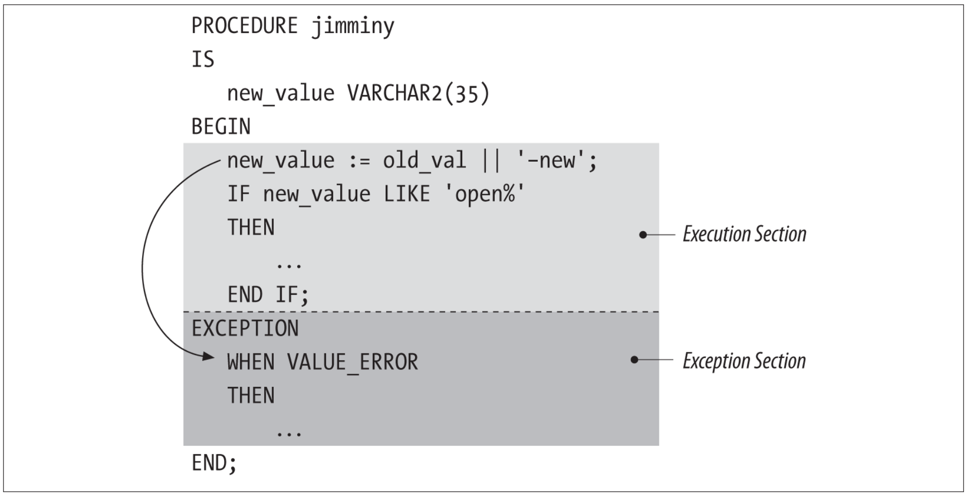
- An error generated by the system (such as "out of memory" or "duplicate value in index")

- An error caused by a user action
- A warning issued by the application to the user

PL/SQL traps and responds to errors using an architecture of exception handlers. The exception handler mechanism allows you to cleanly separate your error-processing code from your executable statements. It also provides an *event-driven* model, as opposed to a linear code model, for processing errors. In other words, no matter how a particular exception is raised, it is handled by the same exception handler in the exception section.

When an error occurs in PL/SQL, whether it's a system error or an application error, an exception is raised. The processing in the current PL/SQL block's execution section halts, and control is transferred to the separate exception section of the current block, if one exists, to handle the exception. You cannot return to that block after you finish handling the exception. Instead, control is passed to the enclosing block, if any.

Figure 6-1 illustrates how control is transferred to the exception section when an exception is raised.



```
PROCEDURE jimminy
IS
   new_value VARCHAR2(35)
BEGIN
   new_value := old_val || '-new';
   IF new_value LIKE 'open%'
   THEN
       ...
   END IF;
EXCEPTION
   WHEN VALUE_ERROR
   THEN
       ...
END;
```

Figure 6-1. Exception-handling architecture

There are, in general, two types of exceptions:

System exception
: An exception that is defined by Oracle and is usually raised by the PL/SQL runtime engine when it detects an error condition. Some system exceptions have names, such as NO_DATA_FOUND, while many others simply have numbers and descriptions.

Programmer-defined exception
: An exception that is defined by the programmer and is therefore specific to the application at hand. You can associate exception names with specific Oracle errors using the EXCEPTION_INIT pragma (a compiler directive, requesting a specific behavior), or you can assign a number and description to that error using RAISE_APPLICATION_ERROR.

The following terms will be used throughout this chapter:

Exception section
: This is the optional section in a PL/SQL block (anonymous block, procedure, function, trigger, or initialization section of a package) that contains one or more "handlers" for exceptions. The structure of an exception section is very similar to that of a CASE statement, which I discussed in Chapter 4.

Raising an exception
: You stop execution of the current PL/SQL block by notifying the runtime engine of an error. The database itself can raise exceptions, or your own code can raise an exception with either the RAISE or RAISE_APPLICATION_ERROR command.

Handle (used as a verb), handler (used as a noun)
: You handle an error by "trapping" it within an exception section. You can then write code in the handler to process that error, which might involve recording the error's occurrence in a log, displaying a message to the user, or propagating an exception out of the current block.

Scope
: This refers to the portion of code (whether in a particular block or for an entire session) in which an exception can be raised. Also, that portion of code for which an exception section can trap and handle exceptions that are raised.

Propagation
: This is the process by which exceptions are passed from one block to its enclosing block if the exception goes unhandled in that block.

Unhandled exception
: An exception is said to go "unhandled" when it propagates without being handled out of the outermost PL/SQL block. Control then passes back to the host execution environment, at which point that environment/program determines how to respond to the exception (roll back the transaction, display an error, ignore it, etc.).

Unnamed or anonymous exception
: This is an exception that has an error code and a description associated with it, but does not have a name that can be used in a RAISE statement or in an exception handler WHEN clause.

Named exception

This refers to an exception that has been given a name, either by Oracle in one of its built-in packages or by a developer. You can also associate a name with an error code through the use of the EXCEPTION_INIT pragma, or leave it defined only by its number (which can be used to both raise and handle the exception).

Defining Exceptions

Before an exception can be raised or handled, it must be defined. Oracle predefines thousands of exceptions, mostly by assigning numbers and messages to those exceptions. Oracle also assigns names to a relative few of these thousands: the most commonly encountered exceptions.

These names are assigned in the STANDARD package (one of two default packages in PL/SQL; DBMS_STANDARD is the other), as well as in other built-in packages such as UTL_FILE and DBMS_SQL. The code Oracle uses to define exceptions like NO_DATA_FOUND is the same that you will write to define or declare your own exceptions. You can do this in two different ways, described in the following sections.

Declaring Named Exceptions

The exceptions that PL/SQL has declared in the STANDARD package (and other built-in packages) cover internal or system-generated errors. Many of the problems a user will encounter (or cause) in an application, however, are specific to that application. Your program might need to trap and handle errors such as "negative balance in account" or "call date cannot be in the past." While different in nature from "division by zero," these errors are still exceptions to normal processing and should be handled gracefully by your program.

One of the most useful aspects of the PL/SQL exception-handling model is that it does not make any structural distinction between internal errors and application-specific errors. Once an exception is raised, it can and should be handled in the exception section, regardless of the type or source of the error.

Of course, to handle an exception, you must have a name for that exception. Because PL/SQL cannot name these exceptions for you (they are specific to your application), you must do so yourself by declaring an exception in the declaration section of your PL/SQL block. You declare an exception by listing the name of the exception you want to raise in your program followed by the keyword EXCEPTION:

```
exception_name EXCEPTION;
```

The following declaration section of the calc_annual_sales procedure contains two programmer-defined exception declarations:

```
PROCEDURE calc_annual_sales
   (company_id_in IN company.company_id%TYPE)
IS
   invalid_company_id   EXCEPTION;
   negative_balance     EXCEPTION;

   duplicate_company    BOOLEAN;
BEGIN
   ... body of executable statements ...
EXCEPTION
  WHEN NO_DATA_FOUND   -- system exception
   THEN
      ...
   WHEN invalid_company_id
   THEN

   WHEN negative_balance
   THEN
      ...
END;
```

The names for exceptions are similar in format to (and "read" just like) Boolean variable names, but can be referenced in only two ways:

- In a RAISE statement in the execution section of the program (to raise the exception), as in:

  ```
  RAISE invalid_company_id;
  ```

- In the WHEN clauses of the exception section (to handle the raised exception), as in:

  ```
  WHEN invalid_company_id THEN
  ```

Associating Exception Names with Error Codes

Oracle has given names to just a handful of exceptions. Thousands of other error conditions within the database are defined by nothing more than an error number and a message. In addition, a developer can raise exceptions using RAISE_APPLICATION_ERROR (covered in "Raising Exceptions" on page 140) that consist of nothing more than an error number (between −20000 and −20999) and an error message.

Exceptions without names are perfectly legitimate, but they can lead to code that is hard to read and maintain. Suppose, for example, that I write a program in which I know the database might raise a date-related error, such as *ORA-01843: not a valid month*. I could write an exception handler to trap that error with code that looks like this:

```
EXCEPTION
   WHEN OTHERS THEN
      IF SQLCODE = -1843 THEN
```

but that is very obscure code, begging for a comment—or some sort of clarity. I can take advantage of the EXCEPTION_INIT statement to make this code's meaning transparent.

SQLCODE is a built-in function that returns the number of the last error raised; it is discussed in "Handling Exceptions" on page 143.

Using EXCEPTION_INIT

EXCEPTION_INIT is a compile-time command or *pragma* used to associate a name with an internal error code. EXCEPTION_INIT instructs the compiler to associate an identifier, declared as an EXCEPTION, with a specific error number. Once you have made that association, you can then raise that exception by name and write an explicit WHEN handler that traps the error.

With EXCEPTION_INIT, I can replace the WHEN clause shown in the previous example with something like this:

```
PROCEDURE my_procedure
IS
   invalid_month EXCEPTION;
   PRAGMA EXCEPTION_INIT (invalid_month, -1843);
BEGIN
   ...
EXCEPTION
   WHEN invalid_month THEN
```

It's more difficult to remember and understand hardcoded error numbers; instead, my code now explains itself.

The pragma EXCEPTION_INIT must appear in the declaration section of a block, and the exception named must have already been defined in that same block, an enclosing block, or a package specification. Here is the syntax in an anonymous block:

```
DECLARE
   exception_name EXCEPTION;
   PRAGMA EXCEPTION_INIT (exception_name, integer);
```

where *exception_name* is the name of an exception and *integer* is a literal integer value, the number of the Oracle error with which you want to associate the named exception. The error number can be any integer value, with these constraints:

- It cannot be –1403 (one of the two error codes for NO_DATA_FOUND). If for some reason you want to associate your own named exception with this error, you need to pass 100 to the EXCEPTION_INIT pragma.

- It cannot be 0 or any positive number besides 100.
- It cannot be a negative number less than −1000000.

Let's look at another example. In the following program code, I declare and associate an exception for this error:

```
ORA-2292 integrity constraint (OWNER.CONSTRAINT) violated  -
         child record found.
```

This error occurs if I try to delete a parent row while it still has existing child rows. (A *child row* is a row with a foreign key reference to the parent table.) The code to declare the exception and associate it with the error code looks like this:

```
PROCEDURE delete_company (company_id_in IN NUMBER)
IS
   /* Declare the exception. */
   still_have_employees EXCEPTION;

   /* Associate the exception name with an error number. */
   PRAGMA EXCEPTION_INIT (still_have_employees, -2292);
BEGIN
   /* Try to delete the company. */
   DELETE FROM company
    WHERE company_id = company_id_in;
EXCEPTION
   /* If child records were found, this exception is raised! */
   WHEN still_have_employees
   THEN
      DBMS_OUTPUT.PUT_LINE
         ('Please delete employees for company first.');
END;
```

Recommended uses of EXCEPTION_INIT

You will find this pragma most useful in two circumstances:

- Giving names to otherwise anonymous system exceptions that you commonly reference in your code—in other words, when Oracle has not predefined a name for the error and you have only the number with which to work.
- Assigning names to the application-specific errors you raise using RAISE_APPLICATION_ERROR (see "Raising Exceptions" on page 140). This allows you to handle such errors by name, rather than simply by number.

In both cases, I recommend that you centralize your usage of EXCEPTION_INIT into packages so that the definitions of exceptions are not scattered throughout your code. Suppose, for example, that I am doing lots of work with dynamic SQL (described in Chapter 16). I might then encounter "invalid column name" errors as I construct my dynamic queries. I don't want to have to remember what the code is for this error, and it would be silly to define my pragmas in 20 different programs. So instead I predefine my own system exceptions in my own dynamic SQL package:

```
CREATE OR REPLACE PACKAGE dynsql
IS
   invalid_table_name EXCEPTION;
      PRAGMA EXCEPTION_INIT (invalid_table_name, -903);
   invalid_identifier EXCEPTION;
      PRAGMA EXCEPTION_INIT (invalid_identifier, -904);
```

and now I can trap for these errors in any program as follows:

```
WHEN dynsql.invalid_identifier THEN ...
```

I suggest that you take this same approach when working with the –20,*NNN* error codes passed to RAISE_APPLICATION_ERROR (described later in this chapter). Avoid hardcoding these literals directly into your application; instead, build (or generate) a package that assigns names to those error numbers. Here is an example of such a package:

```
PACKAGE errnums
IS
   en_too_young CONSTANT NUMBER := -20001;
   exc_too_young EXCEPTION;
   PRAGMA EXCEPTION_INIT (exc_too_young, -20001);

   en_sal_too_low CONSTANT NUMBER := -20002;
   exc_sal_too_low EXCEPTION;
   PRAGMA EXCEPTION_INIT (exc_sal_too_low , -20002);
END errnums;
```

By relying on such a package, I can write code like the following, without embedding the actual error number in the logic:

```
PROCEDURE validate_emp (birthdate_in IN DATE)
IS
   min_years CONSTANT PLS_INTEGER := 18;
BEGIN
   IF ADD_MONTHS (SYSDATE, min_years * 12 * -1) < birthdate_in
   THEN
      RAISE_APPLICATION_ERROR
         (errnums.en_too_young,
         'Employee must be at least ' || min_years || ' old.');
   END IF;
END;
```

About Named System Exceptions

Oracle gives names to a relatively small number of system exceptions by including EXCEPTION_INIT pragma statements in built-in package specifications.

The most important and commonly used set of named exceptions may be found in the STANDARD package in PL/SQL. Because this package is one of the two default packages of PL/SQL, you can reference these exceptions without including the package name as a prefix. So, for instance, if I want to handle the NO_DATA_FOUND exception in my code, I can do so with either of these statements:

```
WHEN NO_DATA_FOUND THEN
WHEN STANDARD.NO_DATA_FOUND THEN
```

You can also find predefined exceptions in other built-in packages, such as DBMS_LOB, the package used to manipulate large objects. Here is an example of one such definition in that package's specification:

```
invalid_argval EXCEPTION;
PRAGMA EXCEPTION_INIT(invalid_argval, -21560);
```

Because DBMS_LOB is not a default package, when I reference this exception, I need to include the package name:

```
WHEN DBMS_LOB.invalid_argval THEN...
```

Many of the STANDARD-based predefined exceptions are listed in Table 6-1, each with its Oracle error number, the value returned by a call to SQLCODE (a built-in function that returns the current error code, described in "Built-in Error Functions" on page 144), and a brief description. In all but one case (100, the ANSI standard error number for NO_DATA_FOUND), the SQLCODE value is the same as the Oracle error code.

Table 6-1. Some of the predefined exceptions in PL/SQL

Name of exception/Oracle error/ SQLCODE	Description
CURSOR_ALREADY_OPEN ORA-6511 SQLCODE = –6511	You tried to OPEN a cursor that was already open. You must CLOSE a cursor before you try to OPEN or re-OPEN it.
DUP_VAL_ON_INDEX ORA-00001 SQLCODE = –1	Your INSERT or UPDATE statement attempted to store duplicate values in a column or columns in a row that is restricted by a unique index.
INVALID_CURSOR ORA-01001 SQLCODE = –1001	You made reference to a cursor that did not exist. This usually happens when you try to FETCH from a cursor or CLOSE a cursor before that cursor is OPENed.
INVALID_NUMBER ORA-01722 SQLCODE = –1722	PL/SQL executed a SQL statement that cannot convert a character string successfully to a number. This exception is different from the VALUE_ERROR exception because it is raised only from within a SQL statement.
LOGIN_DENIED ORA-01017 SQLCODE = –1017	Your program tried to log into the database with an invalid username/password combination. This exception is usually encountered when you embed PL/SQL in a third-generation programming language (3GL).
NO_DATA_FOUND ORA-01403 SQLCODE = +100	This exception is raised in three different scenarios: (1) you executed a SELECT INTO statement (implicit cursor) that returned no rows; (2) you referenced an uninitialized row in a local associative array; or (3) you read past end-of-file with the UTL_FILE package.

Name of exception/Oracle error/ SQLCODE	Description
NOT_LOGGED ON ORA-01012 SQLCODE = –1012	Your program tried to execute a call to the database (usually with a DML statement) before it had logged into the database.
PROGRAM_ERROR ORA-06501 SQLCODE = –6501	PL/SQL has encountered an internal problem. The message text usually also tells you to "Contact Oracle Support."
STORAGE_ERROR ORA-06500 SQLCODE = –6500	Your program ran out of memory, or memory was in some way corrupted.
TIMEOUT_ON_RESOURCE ORA-00051 SQLCODE = –51	A timeout occurred in the database while waiting for a resource.
TOO_MANY_ROWS ORA-01422 SQLCODE = –1422	A SELECT INTO statement returned more than one row. A SELECT INTO must return only one row; if your SQL statement returns more than one row, you should place the SELECT statement in an explicit CURSOR declaration and FETCH from that cursor one row at a time.
TRANSACTION_BACKED_OUT ORA-00061 SQLCODE = –61	The remote part of a transaction was rolled back, either with an explicit ROLLBACK command or as the result of some other action (such as a failed SQL/DML operation on the remote database).
VALUE_ERROR ORA-06502 SQLCODE = –6502	PL/SQL encountered an error having to do with the conversion, truncation, or invalid constraining of numeric and character data. This is a very general and common exception. If this type of error is encountered in a SQL DML statement within a PL/SQL block, then the INVALID_NUMBER exception is raised.
ZERO_DIVIDE ORA-01476 SQLCODE = –1476	Your program tried to divide by zero.

Here is an example of how you might use the exceptions table. Suppose that your program generates an unhandled exception for error ORA-6511. Looking up this error, you find that it is associated with the CURSOR_ALREADY_OPEN exception. Locate the PL/SQL block in which the error occurs and add an exception handler for CURSOR_ALREADY_OPEN, as shown here:

```
EXCEPTION
   WHEN CURSOR_ALREADY_OPEN
   THEN
      CLOSE my_cursor;
END;
```

Of course, you would be even better off analyzing your code to determine proactively which of the predefined exceptions might occur. You could then decide which of those exceptions you want to handle specifically, which should be covered by the WHEN OTHERS clause (discussed later in this chapter), and which would best be left unhandled.

Scope of an Exception

The *scope* of an exception is that portion of the code that is “covered” by that exception. An exception covers a block of code if it can be raised in that block. The following table shows the scope for each of the different kinds of exceptions.

Exception type	Description of scope
Named system exceptions	These exceptions are globally available because they are not declared in or confined to any particular block of code. You can raise and handle a named system exception in any block.
Named programmer-defined exceptions	These exceptions can be raised and handled only in the execution and exception sections of the block in which they are declared (and all nested blocks). If the exception is defined in a package specification, its scope is every program whose owner has EXECUTE privilege on that package.
Anonymous system exceptions	These exceptions can be handled in any PL/SQL exception section via the WHEN OTHERS section. If they are assigned a name, then the scope of that name is the same as that of the named programmer-defined exception.
Anonymous programmer-defined exceptions	These exceptions are defined only in the call to RAISE_APPLICATION_ERROR, and then are passed back to the calling program.

Consider the following example of the exception overdue_balance declared in the procedure check_account. The scope of that exception is the check_account procedure, and nothing else:

```
PROCEDURE check_account (company_id_in IN NUMBER)
IS
   overdue_balance EXCEPTION;
BEGIN
   ... executable statements ...
   LOOP
      ...
      IF ... THEN
         RAISE overdue_balance;
      END IF;
   END LOOP;
EXCEPTION
   WHEN overdue_balance THEN ...
END;
```

I can RAISE the overdue_balance inside the check_account procedure, but I cannot raise that exception from a program that calls check_account. The following anonymous block will generate a compile error, as shown here:

```
DECLARE
   company_id NUMBER := 100;
BEGIN
   check_account (100);
EXCEPTION
   WHEN overdue_balance /* PL/SQL cannot resolve this reference. */
   THEN ...
```

```
END;

PLS-00201: identifier "OVERDUE_BALANCE" must be declared
```

The check_account procedure is a "black box" as far as the anonymous block is concerned. Any identifiers—including exceptions—declared inside check_account are invisible outside of that program.

Raising Exceptions

There are three ways that an exception may be raised in your application:

- The database might raise the exception when it detects an error.
- You might raise an exception with the RAISE statement.
- You might raise an exception with the RAISE_APPLICATION_ERROR built-in procedure.

I've already looked at how the database raises exceptions. Now let's examine the different mechanisms you can use to raise exceptions.

The RAISE Statement

Oracle offers the RAISE statement so that you can, at your discretion, raise a named exception. You can raise an exception of your own or a system exception. The RAISE statement can take one of three forms:

```
RAISE exception_name;
RAISE package_name.exception_name;
RAISE;
```

The first form (without a package name qualifier) can be used to raise an exception you have defined in the current block (or an outer block containing that block) or to raise a system exception defined in the STANDARD package. Here are two examples, first raising a programmer-defined exception:

```
DECLARE
   invalid_id EXCEPTION; -- All IDs must start with the letter 'X'.
   id_value VARCHAR2(30);
BEGIN
   id_value := id_for ('SMITH');
   IF SUBSTR (id_value, 1, 1) != 'X'
   THEN
      RAISE invalid_id;
   END IF;
   ...
END;
```

and then raising a system exception:

```
BEGIN
   IF total_sales = 0
   THEN
      RAISE ZERO_DIVIDE; -- Defined in STANDARD package
   ELSE
      RETURN (sales_percentage_calculation (my_sales, total_sales));
   END IF;
END;
```

The second form does require a package name qualifier. If an exception has been declared inside a package (other than STANDARD) and you are raising that exception outside that package, you must qualify your reference to that exception in your RAISE statement, as in:

```
IF days_overdue (isbn_in, borrower_in) > 365
THEN
   RAISE overdue_pkg.book_is_lost;
END IF;
```

The third form of the RAISE statement does not require an exception name, but can be used only within a WHEN clause of the exception section. Its syntax is simply:

```
RAISE;
```

Use this form when you want to re-raise (or propagate out) the same exception from within an exception handler, as you see here:

```
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      -- Use common package to record all the "context" information,
      -- such as error code, program name, etc.
      errlog.putline (company_id_in);
      -- And now propagate NO_DATA_FOUND unhandled to the enclosing block.
      RAISE;
```

This feature is useful when you want to log the fact that an error occurred but then pass that same error out to the enclosing block. That way, you record where the error occurred in your application but still stop the enclosing block(s) without losing the error information.

Using RAISE_APPLICATION_ERROR

Oracle provides the RAISE_APPLICATION_ERROR procedure (defined in the default DBMS_STANDARD package) to raise application-specific errors in your application. The advantage of using RAISE_APPLICATION_ERROR instead of RAISE (which can also raise an application-specific, explicitly declared exception) is that you can associate an error message with the exception.

When this procedure is run, execution of the current PL/SQL block halts immediately, and any changes made to OUT or IN OUT arguments (if present and without the NO-

COPY hint) will be reversed. Changes made to global data structures, such as packaged variables, and to database objects (by executing an INSERT, UPDATE, MERGE, or DELETE) will *not* be rolled back. You must execute an explicit ROLLBACK to reverse the effect of DML operations.

Here's the header for this procedure (defined in package DBMS_STANDARD):

```
PROCEDURE RAISE_APPLICATION_ERROR (
   num binary_integer,
   msg varchar2,
   keeperrorstack boolean default FALSE);
```

where *num* is the error number and must be a value between −20,999 and −20,000 (just think: Oracle needs all the rest of those negative integers for its *own* exceptions!); *msg* is the error message and must be no more than 2,000 characters in length (any text beyond that limit will be ignored); and *keeperrorstack* indicates whether you want to add the error to any already on the stack (TRUE) or replace the existing errors (the default, FALSE).

> Oracle sets aside the range of −20999 and −20000 for use by its customers, but watch out! Several built-in packages, including DBMS_OUTPUT and DBMS_DESCRIBE, use error numbers between −20005 and −20000. See Oracle's *PL/SQL Packages and Types Reference* for documentation of the use of these error numbers.

Let's take a look at one useful application of this built-in. Suppose that I need to support error messages in different languages for my user community. I create a separate error_table to store all these messages, segregated by the string_language value. I then create a procedure to raise the specified error, grabbing the appropriate error message from the table based on the language used in the current session:

```
/* File on web: raise_by_language.sp */
PROCEDURE raise_by_language (code_in IN PLS_INTEGER)
IS
   l_message error_table.error_string%TYPE;
BEGIN
   SELECT error_string
     INTO l_message
     FROM error_table
    WHERE error_number = code_in
      AND string_language  = USERENV ('LANG');

   RAISE_APPLICATION_ERROR (code_in, l_message);
END;
```

Handling Exceptions

Once an exception is raised, the current PL/SQL block stops its regular execution and transfers control to the exception section. The exception is then either handled by an exception handler in the current PL/SQL block or passed to the enclosing block.

To handle or trap an exception once it is raised, you must write an exception handler for that exception. In your code, your exception handlers must appear after all the executable statements in your program, but before the END statement of the block. The EXCEPTION keyword indicates the start of the exception section and the individual exception handlers:

```
DECLARE
   ... declarations ...
BEGIN
   ... executable statements ...
[ EXCEPTION
   ... exception handlers ... ]
END;
```

The syntax for an exception handler is as follows:

```
WHEN exception_name [ OR exception_name ... ]
THENexecutable statements
```

or:

```
WHEN OTHERS
THEN
   executable statements
```

You can have multiple exception handlers in a single exception section. The exception handlers are structured much like a conditional CASE statement, as shown in the following table.

Property	Description
`EXCEPTION WHEN NO_DATA_FOUND THEN` *`executable_statements1;`*	If the NO_DATA_FOUND exception is raised, then execute the first set of statements.
`WHEN payment_overdue THEN` *`exe cutable_statements2;`*	If the payment is overdue, then execute the second set of statements.
`WHEN OTHERS THEN` *`executable_state ments3;`* `END;`	If any other exception is encountered, then execute the third set of statements.

An exception is handled if its name matches the name of an exception in a WHEN clause. Notice that the WHEN clause traps errors only by exception name, not by error codes. If a match is found, then the executable statements associated with that exception are run. If the exception that has been raised is not handled or does not match any of the named exceptions, the executable statements associated with the WHEN OTHERS clause (if present) will be run. Only one exception handler can catch a particular error.

After the statements for that handler are executed, control passes immediately out of the block.

The WHEN OTHERS clause is optional; if it is not present, then any unhandled exception is immediately propagated back to the enclosing block (if any). The WHEN OTHERS clause must be the last exception handler in the exception section. If you place any other WHEN clauses after WHEN OTHERS, you will receive the following compilation error:

```
PLS-00370: OTHERS handler must be last among the exception handlers of a block
```

Built-in Error Functions

Before exploring the nuances of error handling, let's first review the built-in functions Oracle provides to help you identify, analyze, and respond to errors that occur in your PL/SQL application:

SQLCODE

SQLCODE returns the error code of the most recently raised exception in your block. If there is no error, SQLCODE returns 0. SQLCODE also returns 0 when you call it outside of an exception handler.

The Oracle database maintains a stack of SQLCODE values. Suppose, for example, that function FUNC raises the VALUE_ERROR exception (–6502). Within the exception section of FUNC, you call a procedure PROC that raises DUP_VAL_ON_INDEX (–1). Within the exception section of PROC, SQLCODE returns –1. When control propagates back up to the exception section of FUNC, however, SQLCODE will still return –6502. Run the *sqlcode_test.sql* file (available on the book's website) to see a demonstration of this behavior.

SQLERRM

SQLERRM is a function that returns the error message for a particular error code. If you do not pass an error code to SQLERRM, it returns the error message associated with the value returned by SQLCODE.

If SQLCODE is 0, SQLERRM returns this string:

```
ORA-0000: normal, successful completion
```

If SQLCODE is 1 (the generic user-defined exception error code), SQLERRM returns this string:

```
User-Defined Exception
```

Here is an example of calling SQLERRM to return the error message for a particular code:

```
SQL>  BEGIN
  2     DBMS_OUTPUT.put_line (SQLERRM (-1403));
  3  END;
  4 /
ORA-01403: no data found
```

The maximum-length string that SQLERRM will return is 512 bytes (in some earlier versions of Oracle, only 255 bytes). Because of this restriction, Oracle Corporation recommends that you instead call DBMS_UTILITY.FORMAT_ERROR_STACK to ensure that you see the full error message string (this built-in will not truncate until 2,000 bytes).

The *oracle_error_info.pkg* and *oracle_error_info.tst* files on the book's website provide an example of how you can use SQLERRM to validate error codes.

DBMS_UTILITY.FORMAT_ERROR_STACK

This built-in function, like SQLERRM, returns the message associated with the current error (i.e., the value returned by SQLCODE). It differs from SQLERRM in two ways:

- It will return up to 1,899 characters of error message, thereby avoiding truncation issues.
- You cannot pass an error code number to this function; it cannot be used to return the message for an arbitrary error code.

As a rule, you should call this function inside your exception handler logic to obtain the full error message.

Note that even though the name of the function includes the word *stack*, it doesn't return a stack of errors leading back to the line on which the error was originally raised. That job falls to DBMS_UTILITY.FORMAT_ERROR_ BACKTRACE.

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

Introduced in Oracle Database 10*g*, this function returns a formatted string that displays a stack of programs and line numbers leading back to the line on which the error was originally raised.

This function closed a significant gap in PL/SQL functionality. In Oracle9*i* Database and earlier releases, once you handled an exception inside your PL/SQL block, you were unable to determine the line on which the error had occurred (perhaps the most important piece of information to developers). If you wanted to see this information, you had to allow the exception to go unhandled, at which point the full error backtrace would be displayed on the screen or otherwise presented to the user. This situation is explored in more detail in the following section.

DBMS_UTILITY.FORMAT_CALL_STACK

This function returns a formatted string showing the execution call stack inside your PL/SQL application. Its usefulness is not restricted to error management; you will also find it handy for tracing the execution of your code. This function is explored in more detail in Chapter 20.

In Oracle Database 12*c*, Oracle introduced the UTL_CALL_STACK package, which also provides you with access to the call stack, error stack, and backtrace information. This package is explored in Chapter 20.

More on DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

You should call the DBMS_UTILITY.FORMAT_ERROR_BACKTRACE function in your exception handler. It displays the execution stack at the point where an exception was raised. Thus, you can call DBMS_UTILITY.FORMAT_ERROR_BACKTRACE within an exception section at the top level of your stack and still find out where the error was raised deep within the call stack.

Consider the following scenario: I define a procedure proc3, which calls proc2, which in turn calls proc1. The proc1 procedure raises an exception:

```
CREATE OR REPLACE PROCEDURE proc1 IS
BEGIN
   DBMS_OUTPUT.put_line ('running proc1');
   RAISE NO_DATA_FOUND;
END;
/

CREATE OR REPLACE PROCEDURE proc2 IS
   l_str VARCHAR2 (30) := 'calling proc1';
BEGIN
   DBMS_OUTPUT.put_line (l_str);
   proc1;
END;
/

CREATE OR REPLACE PROCEDURE proc3 IS
BEGIN
   DBMS_OUTPUT.put_line ('calling proc2');
   proc2;
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line ('Error stack at top level:');
      DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_backtrace);
END;
/
```

The only program with an exception handler is the outermost program, proc3. I have placed a call to the backtrace function in proc3's WHEN OTHERS handler. When I run this procedure, I see the following results:

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
  2     DBMS_OUTPUT.put_line ('Proc3 -> Proc2 -> Proc1 backtrace');
  3     proc3;
  4  END;
  5  /

Proc3 -> Proc2 -> Proc1 backtrace
calling proc2
calling proc1
running proc1
Error stack at top level:
ORA-06512: at "SCOTT.PROC1", line 4
ORA-06512: at "SCOTT.PROC2", line 5
ORA-06512: at "SCOTT.PROC3", line 4
```

As you can see, the backtrace function shows at the top of its stack the line in proc1 on which the error was originally raised.

Often, an exception occurs deep within the execution stack. If you want that exception to propagate all the way to the outermost PL/SQL block, it may have to be re-raised within each exception handler in the stack of blocks. DBMS_UTILITY.FORMAT_ERROR_BACKTRACE shows the trace of execution back to the last RAISE in one's session. As soon as you issue a RAISE of a particular exception or re-raise the current exception, you restart the stack that the DBMS_UTILITY.FORMAT_ERROR_BACKTRACE function produces. This means that if you want to take advantage of this function, you should take one of the following two approaches:

- Call the function in the exception section of the block in which the error was raised. This way you have (and can log) that critical line number, even if the exception is re-raised further up in the stack.
- Avoid exception handlers in intermediate programs in your stack, and call the function in the exception section of the outermost program in your stack.

Just the line number, please

In a real-world application, the error backtrace could be very long. Generally, the person doing the debugging or support doesn't really want to have to deal with the entire stack, and is mostly going to be interested only in that topmost entry. The developer of the application might even want to display that critical information so that the user can immediately and accurately report the problem to the support team.

In this case, it is necessary to parse the backtrace string and retrieve just the topmost entry. I built a utility to do this called the BT package; you can download it from the book's website. In this package, I provide a simple, clean interface as follows:

```
/* File on web: bt.pkg */
PACKAGE bt
IS
  TYPE error_rt IS RECORD (
    program_owner all_objects.owner%TYPE
  , program_name all_objects.object_name%TYPE
  , line_number PLS_INTEGER
  );

  FUNCTION info (backtrace_in IN VARCHAR2)
    RETURN error_rt;

  PROCEDURE show_info (backtrace_in IN VARCHAR2);
END bt;
```

The record type, error_rt, contains a separate field for each element of the backtrace that I want to retrieve (owner of the program unit, name of the program unit, and line number within that program). Then, instead of calling and parsing the backtrace function in each exception section, I can call the bt.info function and report on the specifics of the error.

Useful applications of SQLERRM

While it is true that you should use DBMS_UTILITY.FORMAT_ERROR_STACK in place of SQLERRM, that doesn't mean SQLERRM is totally irrelevant. In fact, you can use it to answer the following questions:

- Is a particular number a valid Oracle error?
- What is the error message corresponding to an error code?

As mentioned earlier in this chapter, SQLERRM will return the error message for an error code. If, however, you pass SQLERRM a code that is not valid, it does not raise an exception. Instead, it returns a string in one of the following two forms:

- If the number is negative:

  ```
  ORA-NNNNN: Message NNNNN not found;  product=RDBMS; facility=ORA
  ```

- If the number is positive or less than –65535:

  ```
  -N: non-ORACLE exception
  ```

You can use these facts to build functions to neatly return information about whatever code you are currently working with. Here is the specification of a package with such programs:

```
/* File on web: oracle_error_info.pkg */
PACKAGE oracle_error_info
IS
   FUNCTION is_app_error (code_in IN INTEGER)
      RETURN BOOLEAN;

   FUNCTION is_valid_oracle_error (
      code_in            IN   INTEGER
```

```
     , app_errors_ok_in   IN   BOOLEAN DEFAULT TRUE
     , user_error_ok_in   IN   BOOLEAN DEFAULT TRUE
    )
       RETURN BOOLEAN;

    PROCEDURE validate_oracle_error (
       code_in            IN       INTEGER
     , message_out        OUT      VARCHAR2
     , is_valid_out       OUT      BOOLEAN
     , app_errors_ok_in   IN       BOOLEAN DEFAULT TRUE
     , user_error_ok_in   IN       BOOLEAN DEFAULT TRUE
    );
END oracle_error_info;
```

You will find the complete implementation on the book's website.

Combining Multiple Exceptions in a Single Handler

You can, within a single WHEN clause, combine multiple exceptions together with an OR operator, just as you would combine multiple Boolean expressions:

```
WHEN invalid_company_id OR negative_balance
THEN
```

You can also combine application and system exception names in a single handler:

```
WHEN balance_too_low OR ZERO_DIVIDE OR DBMS_LDAP.INVALID_SESSION
THEN
```

You cannot, however, use the AND operator because only one exception can be raised at a time.

Unhandled Exceptions

If an exception is raised in your program, and it is not handled by an exception section in either the current or enclosing PL/SQL blocks, that exception is *unhandled*. PL/SQL returns the error that raised the unhandled exception all the way back to the application environment from which PL/SQL was run. That environment (a tool like SQL*Plus, Oracle Forms, or a Java program) then takes an action appropriate to the situation; in the case of SQL*Plus, a ROLLBACK of any DML changes from within that top-level block's logic is automatically performed.

One key decision to make about your application architecture is whether you want to allow unhandled exceptions to occur at all. They are handled differently by different frontends, and in some cases none too gracefully. If your PL/SQL programs are being called from a non-PL/SQL environment, you may want to design your outermost blocks or programs to do the following:

- Trap any exception that might have propagated out to that point.

- Log the error so that a developer can analyze what might be the cause of the problem.
- Pass back a status code, description, and any other information needed by the host environment to determine the appropriate action to take.

Propagation of Unhandled Exceptions

The scope rules for exceptions determine the block in which an exception can be raised. The rules for exception propagation address the way in which an exception is handled after it is raised.

When an exception is raised, PL/SQL looks for an exception handler in the current block (anonymous block, procedure, or function) of the exception. If it does not find a match, then PL/SQL propagates the exception to the enclosing block of that current block. PL/SQL then attempts to handle the exception by raising it once more in the enclosing block. It continues to do this in each successive enclosing block until there are no more blocks in which to raise the exception (see Figure 6-2). When all blocks are exhausted, PL/SQL returns an unhandled exception to the application environment that executed the outermost PL/SQL block. An unhandled exception halts the execution of the host program.

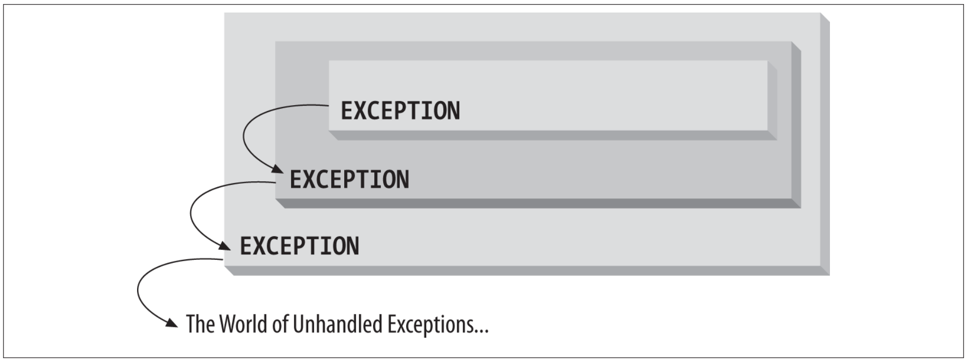


Figure 6-2. Propagation of an exception through nested blocks

Losing exception information

The architecture of PL/SQL exception handling leads to an odd situation regarding local, programmer-defined exceptions: you can lose crucial information (what error occurred?) unless you are careful.

Consider the following situation. I declare an exception as follows:

```
BEGIN
    <<local_block>>
    DECLARE
```

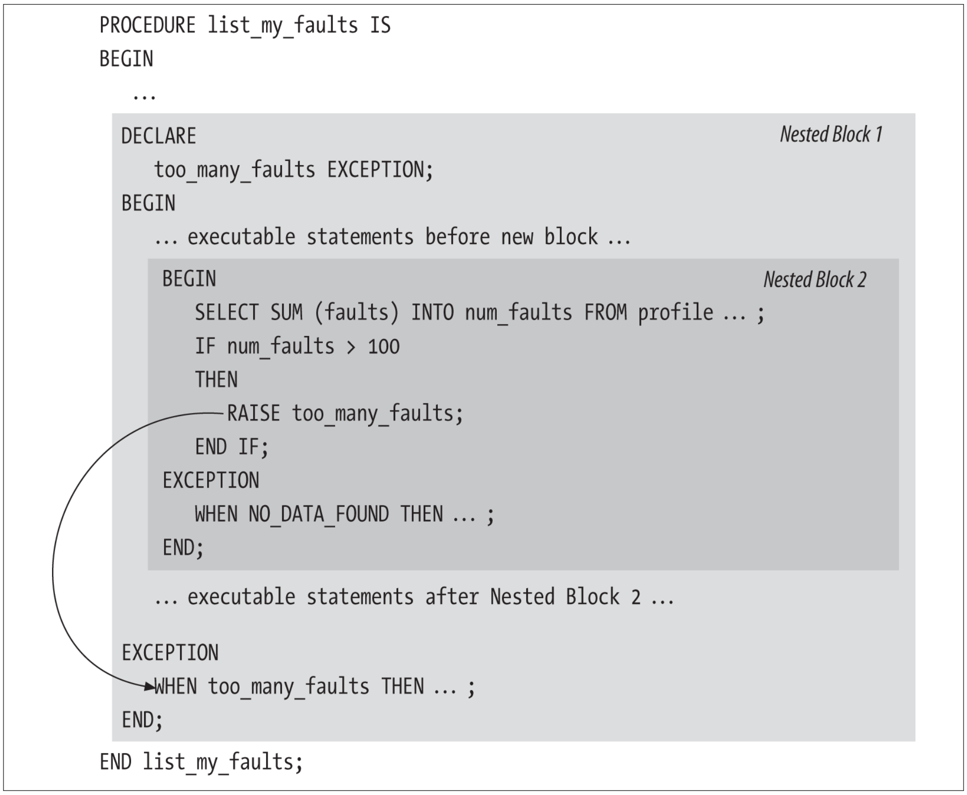
```
        case_is_not_made EXCEPTION;
    BEGIN
        ...
    END local_block;
```

but neglect to include an exception section. The scope of the case_is_not_made exception is inside local_block's execution and exception sections. If the exception is not handled there and instead propagates to the enclosing block, then there is no way to know that the case_is_not_made exception was raised. You really don't know *which* error was raised, only that some error was raised. That's because all user-defined exceptions have an error code of 1 and an error message of "User Defined Exception"—unless you use the EXCEPTION_INIT pragma to associate a different number with that declared exception, and use RAISE_APPLICATION_ERROR to associate it with a different error message.

As a consequence, when you are working with locally defined (and raised) exceptions, you should include exception handlers specifically for those errors by name.

Examples of exception propagation

Let's look at a few examples of how exceptions propagate through enclosing blocks. Figure 6-3 shows how the exception raised in the inner block, too_many_faults, is handled by the next enclosing block. The innermost block has an exception section, so PL/SQL first checks to see if too_many_faults is handled in this section. Because it is not handled, PL/SQL closes that block and raises the too_many_faults exception in the enclosing block, Nested Block 1. Control immediately passes to the exception section of Nested Block 1. (The executable statements after Nested Block 2 are not executed.) PL/SQL scans the exception handlers and finds that too_many_faults is handled in this block, so the code for that handler is executed, and control passes back to the main list_my_faults procedure.



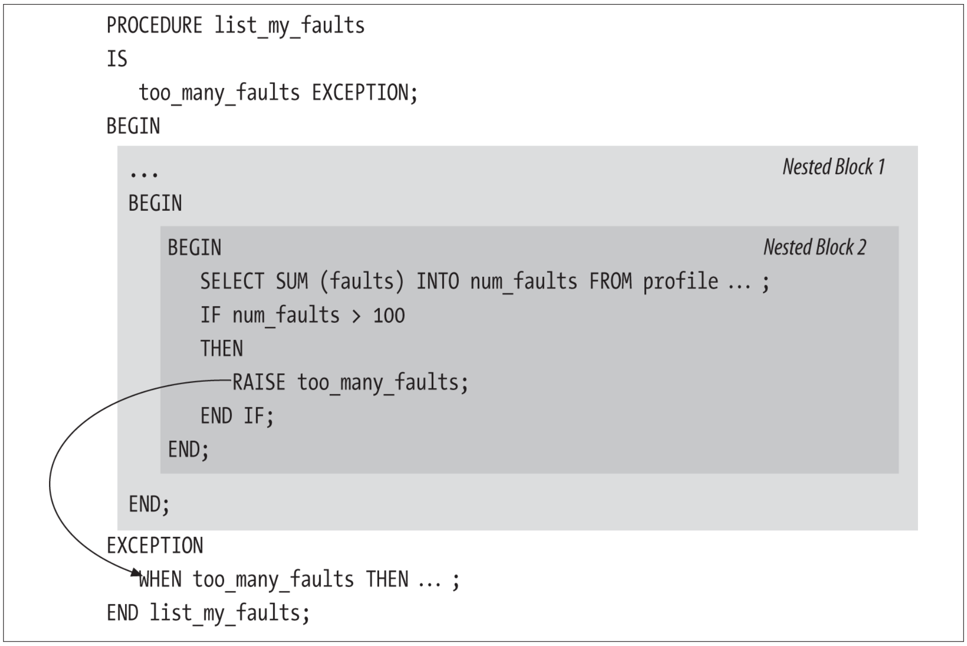
```
PROCEDURE list_my_faults IS
BEGIN
  ...
  DECLARE
     too_many_faults EXCEPTION;
  BEGIN
     ... executable statements before new block ...
     BEGIN
        SELECT SUM (faults) INTO num_faults FROM profile ... ;
        IF num_faults > 100
        THEN
           RAISE too_many_faults;
        END IF;
     EXCEPTION
        WHEN NO_DATA_FOUND THEN ... ;
     END;
     ... executable statements after Nested Block 2 ...

  EXCEPTION
     WHEN too_many_faults THEN ... ;
  END;
END list_my_faults;
```

Figure 6-3. Propagation of exception handling to first nested block

Notice that if the NO_DATA_FOUND exception had been raised in the innermost block (Nested Block 2), then the exception section for Nested Block 2 would have handled the exception. Then control would have passed back to Nested Block 1, and the executable statements that come after Nested Block 2 would have been executed.

In Figure 6-4, the exception raised in the inner block is handled by the outermost block. The outermost block is the only one with an exception section, so when Nested Block 2 raises the too_many_faults exception, PL/SQL terminates execution of that block and raises that exception in the enclosing block, Nested Block 1. Again, this block has no exception section, so PL/SQL immediately terminates Nested Block 1 and passes control to the outermost block, the list_my_faults procedure. This procedure does have an exception section, so PL/SQL scans the exception handlers, finds a match for too_many_faults, executes the code for that handler, and then returns control to whatever program called list_my_faults.



```
PROCEDURE list_my_faults
IS
   too_many_faults EXCEPTION;
BEGIN
  ...
  BEGIN
     BEGIN
        SELECT SUM (faults) INTO num_faults FROM profile ... ;
        IF num_faults > 100
        THEN
           RAISE too_many_faults;
        END IF;
     END;
  END;
EXCEPTION
   WHEN too_many_faults THEN ... ;
END list_my_faults;
```

Figure 6-4. Exception raised in nested block handled by outermost block

Continuing Past Exceptions

When an exception is raised in a PL/SQL block, normal execution is halted and control is transferred to the exception section. You can never return to the execution section once an exception is raised in that block. In some cases, however, the ability to continue past exceptions is exactly the desired behavior.

Consider the following scenario: I need to write a procedure that performs a series of DML statements against a variety of tables (delete from one table, update another, insert into a final table). My first pass at writing this procedure might produce code like the following:

```
PROCEDURE change_data IS
BEGIN
   DELETE FROM employees WHERE ... ;
   UPDATE company SET ... ;
   INSERT INTO company_history SELECT * FROM company WHERE ... ;
END;
```

This procedure certainly contains all the appropriate DML statements. But one of the requirements for this program is that, although these statements are executed in sequence, they are logically independent of each other. In other words, even if the DELETE fails, I want to go on and perform the UPDATE and INSERT.

With the current version of change_data, I can't make sure that all three DML statements will at least be attempted. If an exception is raised from the DELETE, for example, the entire program's execution will halt, and control will be passed to the exception section, if there is one. The remaining SQL statements won't be executed.

How can I get the exception to be raised and handled without terminating the program as a whole? The solution is to place the DELETE within its own PL/SQL block. Consider this next version of the change_data program:

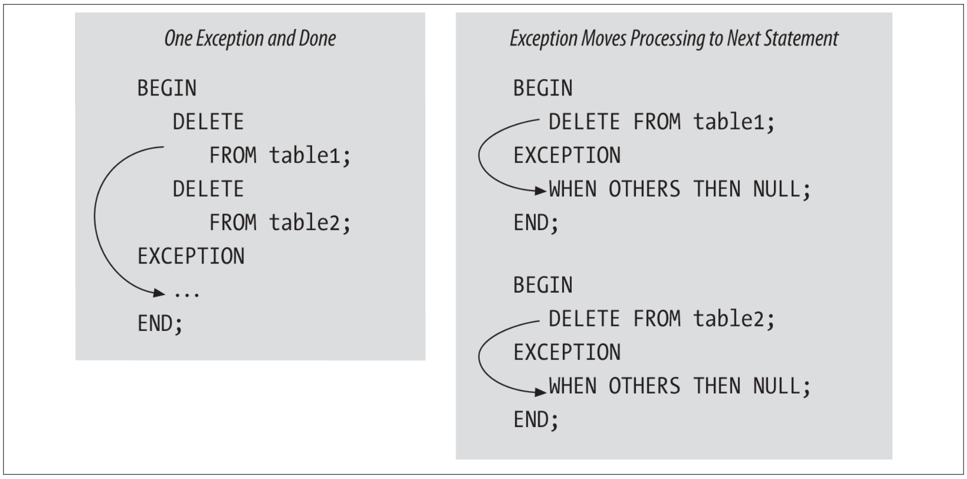
```
PROCEDURE change_data IS
BEGIN
   BEGIN
      DELETE FROM employees WHERE ... ;
   EXCEPTION
      WHEN OTHERS THEN log_error;
   END;

   BEGIN
      UPDATE company SET ... ;
   EXCEPTION
      WHEN OTHERS THEN log_error;
   END;

   BEGIN
      INSERT INTO company_history SELECT * FROM company WHERE ... ;
   EXCEPTION
      WHEN OTHERS THEN log_error;
   END;
END;
```

With this new format, if the DELETE raises an exception, control is immediately passed to the exception section. But what a difference! Because the DELETE statement is now in its own block, it can have its own exception section. The WHEN OTHERS clause in that section smoothly handles the error by logging its occurrence, *without re-raising this or any other error*. Control is then passed out of the DELETE's block and back to the enclosing change_data procedure. Since there is no longer an "active" exception, execution continues in this enclosing block to the next statement in the procedure. A new anonymous block is then entered for the UPDATE statement. If the UPDATE statement fails, the WHEN OTHERS clause in the UPDATE's own exception section traps the problem and returns control to change_data, which blithely moves on to the INSERT statement (also contained in its very own block).

Figure 6-5 shows this process for two sequential DELETE statements.



```
BEGIN
   DELETE
      FROM table1;
   DELETE
      FROM table2;
EXCEPTION
   ...
END;
```

```
BEGIN
   DELETE FROM table1;
EXCEPTION
   WHEN OTHERS THEN NULL;
END;

BEGIN
   DELETE FROM table2;
EXCEPTION
   WHEN OTHERS THEN NULL;
END;
```

Figure 6-5. Sequential DELETEs, using two different approaches to scope

To summarize: an exception raised in the executable section will always be handled in the current block—if there is a matching handler present. You can create a virtual block around any statement(s) by prefacing it with a BEGIN and following it with an EXCEPTION section and an END statement. In this way you can control the scope of failure caused by an exception by establishing buffers of anonymous blocks in your code.

You can also take this strategy a step further and move the code you want to isolate into separate procedures or functions. Of course, these named PL/SQL blocks may also have their own exception sections and will offer the same protection from total failure. One key advantage of using procedures and functions is that you hide all the BEGIN-EXCEPTION-END statements from the mainline program. The program is then easier to read, understand, maintain, and reuse in multiple contexts.

There are other ways to continue past a DML exception. You can also use SAVE EXCEPTIONS with FORALL and LOG ERRORS in association with DBMS_ERRORLOG to continue past exceptions raised by DML.

Writing WHEN OTHERS Handling Code

You include the WHEN OTHERS clause in the exception section to trap any otherwise unhandled exceptions. Because you have not explicitly handled any specific exceptions here, you will very likely want to take advantage of the built-in error functions, such as SQLCODE and DBMS_UTILITY.FORMAT_ERROR_STACK, to give you information about the error that has occurred.

Combined with WHEN OTHERS, SQLCODE provides a way for you to handle different specific exceptions without having to use the EXCEPTION_INIT pragma. In the next example, I trap two parent-child exceptions, –1 and –2292, and then take an action appropriate to each situation:

```
PROCEDURE add_company (
   id_in      IN company.ID%TYPE
 , name_in    IN company.name%TYPE
 , type_id_in IN company.type_id%TYPE
)
IS
BEGIN
   INSERT INTO company (ID, name, type_id)
        VALUES (id_in, name_in, type_id_in);
EXCEPTION
   WHEN OTHERS
   THEN
      /*
      || Anonymous block inside the exception handler lets me declare
      || local variables to hold the error code information.
      */
      DECLARE
         l_errcode PLS_INTEGER := SQLCODE;
      BEGIN
         CASE l_errcode
         WHEN –1 THEN
            -- Duplicate value for unique index. Either a repeat of the
            -- primary key or name. Display problem and re-raise.
            DBMS_OUTPUT.put_line
                              (   'Company ID or name already in use. ID = '
                               || TO_CHAR (id_in)
                               || ' name = '
                               || name_in
                              );
            RAISE;
         WHEN –2291 THEN
            -- Parent key not found for type. Display problem and re-raise.
            DBMS_OUTPUT.put_line (
               'Invalid company type ID: ' || TO_CHAR (type_id_in));
            RAISE;
         ELSE
            RAISE;
         END CASE;
      END; -- End of anonymous block.
END add_company;
```

You should use WHEN OTHERS with care, because it can easily “swallow up” errors and hide them from the outer blocks and the user. Specifically, watch out for WHEN OTHER handlers that do not re-raise the current exception or raise some other exception in its place. If WHEN OTHERS does not propagate out an exception, then the outer blocks of your application will never know that an error occurred.

Oracle Database 11*g* offers a new warning to help you identify programs that may be ignoring or swallowing up errors:

```
PLW-06009: procedure "string" OTHERS handler does not end in RAISE or RAISE_
APPLICATION_ERROR
```

Here is an example of using this warning:

```
/* File on web: plw6009.sql */
SQL> ALTER SESSION SET plsql_warnings = 'enable:all'
  2  /

SQL> CREATE OR REPLACE PROCEDURE plw6009_demo
  2  AS
  3  BEGIN
  4     DBMS_OUTPUT.put_line ('I am here!');
  5     RAISE NO_DATA_FOUND;
  6  EXCEPTION
  7     WHEN OTHERS
  8     THEN
  9        NULL;
 10  END plw6009_demo;
 11  /

SP2-0804: Procedure created with compilation warnings

SQL> SHOW ERRORS
Errors for PROCEDURE PLW6009_DEMO:

LINE/COL ERROR
-------- -----------------------------------------------------------------
7/9      PLW-06009: procedure "PLW6009_DEMO" OTHERS handler does not end
         in RAISE or RAISE_APPLICATION_ERROR
```

Building an Effective Error Management Architecture

PL/SQL error raising and handling mechanisms are powerful and flexible, but they have some drawbacks that can present challenges to any development team that wants to implement a robust, consistent, informative architecture for error management.

Here are the some of the challenges you will encounter:

- The EXCEPTION is an odd kind of structure in PL/SQL. A variable declared to be EXCEPTION can only be raised and handled. It has at most two characteristics: an error code and an error message. You cannot pass an exception as an argument to a program; you cannot associate other attributes with an exception.
- It is very difficult to reuse exception-handling code. Directly related to the previous challenge is another fact: you cannot pass an exception as an argument; you end

up cutting and pasting handler code, which is certainly not an optimal way to write programs.

- There is no formal way to specify which exceptions a program may raise. With Java, on the other hand, this information becomes part of the specification of the program. The consequence is that you must look inside the program implementation to see what might be raised—or hope for the best.
- Oracle does not provide any way for you to organize and categorize your application-specific exceptions. It simply sets aside (for the most part) the 1,000 error codes between −20,999 and −20,000. You are left to manage those values.

Let's figure out how we can best meet most of these challenges.

Decide on Your Error Management Strategy

It is extremely important that you establish a consistent strategy and architecture for error handling in your application before you write any code. To do that, you must answer questions like these:

- How and when do I log errors so that they can be reviewed and corrected? Should I write information to a file, to a database table, and/or to the screen?
- How and when do I report the occurrence of errors back to the user? How much information should the user see and have to keep track of? How do I transform often obscure database error messages into text that is understandable to my users?

Linked tightly to these very high-level questions are more concrete issues, such as:

- Should I include an exception-handling section in every one of my PL/SQL blocks?
- Should I have an exception-handling section only in the top-level or outermost blocks?
- How should I manage my transactions when errors occur?

Part of the complexity of exception handling is that there is no single right answer to any of these questions. It depends at least in part on the application architecture and the way it is used (batch process versus user-driven transactions, for example). However you answer these questions for your application, I strongly suggest that you "codify" the strategy and rules for error handling within a standardized package. I address this topic in "Use Standardized Error Management Programs" on page 163.

Here are some general principles you may want to consider:

- When an error occurs in your code, obtain as much information as possible about the context in which the error was raised. You are better off with more information than you really need, rather than with less. You can then propagate the exception to outer blocks, picking up more information as you go.

- Avoid hiding errors with handlers that look like WHEN *error* THEN NULL; (or, even worse, WHEN OTHERS THEN NULL;). There may be a good reason for you to write code like this, but make sure it is really what you want and document the usage so that others will be aware of it.
- Rely on the default error mechanisms of PL/SQL whenever possible. Avoid writing programs that return status codes to the host environment or calling blocks. The only time you will want to use status codes is if the host environment cannot gracefully handle Oracle errors (in which case, you might want to consider switching your host environment!).

Standardize Handling of Different Types of Exceptions

An exception is an exception is an exception? Not really. Some exceptions, for example, indicate that the database is having very severe, low-level problems (such as ORA-00600). Other exceptions, like NO_DATA_FOUND, happen so routinely that we don't even really necessarily think of them as *errors*, but more as a conditional branching of logic ("If the row doesn't exist, then do this..."). Do these distinctions really matter? I think so, and Bryn Llewellyn, PL/SQL Product Manager as of the writing of this book, taught me a very useful way to categorize exceptions:

Deliberate
: The code architecture itself deliberately relies upon an exception in the way it works. This means you must (well, *should*) anticipate and code for this exception. An example is UTL_FILE.GET_LINE.

Unfortunate
: This is an error, but one that is to be expected and may not even indicate that a problem has occurred. An example is a SELECT INTO statement that raises NO_DATA_FOUND.

Unexpected
: This is a "hard" error indicating a problem in the application. An example is a SELECT INTO statement that is supposed to return a row for a given primary key, but instead raises TOO_MANY ROWS.

Let's take a closer look at the examples given here for each of these exception categories. Then I will discuss how knowing about these categories can and should be useful to you.

Deliberate exceptions

PL/SQL developers can use UTL_FILE.GET_LINE to read the contents of a file, one line at a time. When GET_LINE reads past the end of a file, it raises NO_DATA_FOUND. That's just the way it works. So, if I want to read everything from a file and "do stuff," my program might look like this:

```
PROCEDURE read_file_and_do_stuff (
   dir_in IN VARCHAR2, file_in IN VARCHAR2
)
IS
   l_file   UTL_FILE.file_type;
   l_line   VARCHAR2 (32767);
BEGIN
   l_file := UTL_FILE.fopen (dir_in, file_in, 'R', max_linesize => 32767);

   LOOP
      UTL_FILE.get_line (l_file, l_line);
      do_stuff;
   END LOOP;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      UTL_FILE.fclose (l_file);
      more_stuff_here;
END;
```

You may notice something a bit strange about my loop; it has no EXIT statement. Also, I am running more application logic (more_stuff_here) in the exception section. I can rewrite my loop as follows:

```
LOOP
   BEGIN
      UTL_FILE.get_line (l_file, l_line);
      do_stuff;
   EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
         EXIT;
   END;
   END LOOP;

   UTL_FILE.flcose (l_file);
   more_stuff_here;
```

Now I have an EXIT statement in my loop, but that sure is some awkward code.

This is the kind of thing you need to do when you work with code that deliberately raises an exception as a part of its architecture. You'll find more in the next few sections about what I think you should do about this.

Unfortunate and unexpected exceptions

I will cover these together because the two examples (NO_DATA_FOUND and TOO_MANY_ROWS) are tightly linked. Suppose I need to write a function to return the full name of an employee (last comma first) for a particular primary key value.

I could write it most simply as follows:

```
FUNCTION fullname (
   employee_id_in IN employees.employee_id%TYPE
)
   RETURN VARCHAR2
IS
   retval   VARCHAR2 (32767);
BEGIN
   SELECT last_name || ',' || first_name
     INTO retval
     FROM employees
    WHERE employee_id = employee_id_in;

   RETURN retval;
END fullname;
```

If I call this program with an employee ID that is not in the table, the database will raise the NO_DATA_FOUND exception. If I call this program with an employee ID that is found in more than one row in the table, the database will raise the TOO_MANY_ROWS exception.

One query, two different exceptions—should I treat them the same way? Perhaps not. Do these two exceptions truly reflect similar kinds of problems? Let's see:

NO_DATA_FOUND
: With this exception I didn't find a match. That *could* be a serious problem, but that's not necessarily the case. Perhaps I actually expect that most of the time I will not get a match, and therefore will simply insert a new employee. It is, shall we say, *unfortunate* that the exception was raised, but in this case it is not even an error.

TOO_MANY_ROWS
: With this exception I have a serious problem on my hands: something has gone wrong with my primary key constraint. I can't think of a circumstance in which this would be considered OK or simply "unfortunate." No, it is time to stop the program and call attention to this very unexpected, "hard" error.

How to benefit from this categorization

I hope you agree that this characterization sounds useful. I suggest that when you are about to build a new application, you decide as much as possible the standard approach you (and everyone else on the team) will take for each type of exception. Then, as you encounter (need to handle or write in anticipation of) an exception, decide into which category it falls, and then apply the already decided upon approach. In this way, you will all write your code in a more consistent and productive manner.

Here are my guidelines for dealing with the three types of exceptions:

Deliberate
: You will need to write code in anticipation of these exceptions. The critical best practice in this case is to *avoid putting application logic in the exception section*. The exception section should contain only code needed to deal with the error: logging the error data, re-raising the exception, etc. Programmers don't expect application-specific logic there, which means that it will be much harder to understand and maintain.

Unfortunate
: If there are circumstances under which a user of the code that raises these exceptions would not interpret the situation as an *error*, then don't propagate these exceptions out unhandled. Instead, return a value or status flag that indicates an exception was raised. You then leave it up to the user of the program to decide if that program should terminate with an error. Better yet, why not let the caller of your program tell it whether or not to raise an exception, and if not, what value should be passed to indicate that the exception occurred?

Unexpected
: Now we are down to the hard stuff. All unexpected errors should be logged, recording as much of the application context as possible to help understand why the errors occurred. The program should then terminate with an unhandled exception (usually the same one) that was raised within the program, which can be done with the RAISE statement, forcing the calling program to stop and deal with the error.

Organize Use of Application-Specific Error Codes

When you use RAISE_APPLICATION_ERROR to raise application-specific errors, it is entirely up to you to manage the error codes and messages. This can get tricky and messy ("Gee, which number should I use? Well, I doubt that anyone will be using −20774!").

To help manage your error codes and provide a consistent interface with which developers can handle server errors, consider building a table to store all the −20,*NNN* error numbers you use, along with their associated exception names and error messages. Developers can then view these already defined errors via a screen and choose the one that fits their situation. See the *msginfo.sql* file on the book's website for one such example of a table, along with code that will generate a package containing declarations of each of the "registered" exceptions.

Another approach you can take is to avoid the −20,*NNN* range entirely for application-specific errors. Why not use positive numbers instead? Oracle uses only 1 and 100 on the positive side of the integer range. While it is *possible* that Oracle will, over time, use other positive numbers, it is very unlikely. That leaves an awful lot of error codes for us to use.

I took this approach when designing the Quest Error Manager (QEM), a freeware error management utility. With the Quest Error Manager, you can define your own errors in a special repository table. You can define an error by name and/or error code. The error codes can be negative or positive. If the error code is positive, then when you raise that exception, QEM uses RAISE_APPLICATION_ERROR to raise a generic exception (usually −20,000). The information about the current application error code is embedded in the error message, which can then be decoded by the receiving program.

You can also see a simpler implementation of this approach in the general error manager package, *errpkg.pkg*, which is described in the next section.

Use Standardized Error Management Programs

Robust and consistent error handling is an absolutely crucial element of a properly constructed application. This consistency is important for two very different audiences: the user and the developer. If presented with easy-to-understand, well-formatted information when an error occurs, the user will be able to report that error more effectively to the support team and will feel more comfortable using the application. If the application handles and logs errors in the same way throughout, the support and maintenance programmers will be able to fix and enhance the code much more easily.

Sounds like a sensible approach, doesn't it? Unfortunately, and especially in development teams of more than a handful of people, the end result of exception handling is usually very different from what I just described. A more common practice is that each developer strikes out on his own path, following different principles, writing to different kinds of logs, and so on. Without standardization, debugging and maintenance become a nightmare. Here's an example of the kind of code that typically results:

```
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
       v_msg := 'No company for id '||TO_CHAR (v_id);
       v_err := SQLCODE;
       v_prog := 'fixdebt';
       INSERT INTO errlog VALUES
          (v_err,v_msg,v_prog,SYSDATE,USER);

    WHEN OTHERS
    THEN
       v_err := SQLCODE;
       v_msg := SQLERRM;
       v_prog := 'fixdebt';
       INSERT INTO errlog VALUES
```

```
            (v_err,v_msg,v_prog,SYSDATE,USER);
         RAISE;
```

At first glance, this code might seem quite sensible, and in fact explains itself clearly:

> If I don't find a company for this ID, grab the SQLCODE value, set the program name and message, and write a row to the log table. Then allow the enclosing block to continue (it's not a very severe error in this case). If any other error occurs, grab the error code and message, set the program name, write a row to the log table, and then propagate out the same exception, causing the enclosing block to stop (I don't know how severe the error is).

So what's wrong with all that? The mere fact that I can actually explain everything that is going on is an indication of the problem. I have exposed and hardcoded all the steps I take to get the job done. The result is that (1) I write a lot of code, and (2) if anything changes, I have to change a lot of code. Just to give you one example, notice that I am writing to a database table for my log. This means that the log entry has become a part of my logical transaction. If I need to roll back that transaction, I lose my error log.

There are several ways to correct this problem—for example, I could write to a file or use autonomous transactions to save my error log without affecting my main transaction. The problem is that, with the way I have written the preceding code, I have to apply my correction in potentially hundreds of different programs.

Now consider a rewrite of this same exception section using a standardized package:

```
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      errpkg.record_and_continue (
         SQLCODE, 'No company for id ' || TO_CHAR (v_id));

   WHEN OTHERS
   THEN
      errpkg.record_and_stop;
END;
```

My error-handling package hides all the implementation details; I simply decide which of the handler procedures I want to use by viewing the specification of the package. If I want to record the error and then continue, I call record_and_continue. If I want to record and then stop, clearly I want to use record_and_stop. How does it record the error? How does it stop the enclosing block (i.e., how does it propagate the exception)? I don't know, and I don't care. Whatever it does, it does it according to the standards defined for my application.

All I know is that I can now spend more time building the interesting elements of my application, rather than worrying over the tedious, low-level administrivia.

The *errpkg.pkg* file available on the book's website contains a prototype of such a standardized error-handling package. You will want to review and complete its implementation before using it in your application, but it will give you a very clear sense of how to construct such a utility.

Alternatively, you can take advantage of a much more complete error management utility (also free): the Quest Error Manager mentioned in the previous section. The most important concept underlying my approach with QEM is that you trap and log information about *instances* of errors, and not just the Oracle errors themselves. QEM consists of a PL/SQL package and four underlying tables that store information about errors that occur in an application.

Work with Your Own Exception "Objects"

Oracle's implementation of the EXCEPTION datatype has some limitations, as described earlier. An exception consists of an identifier (a name) with which you can associate a number and a message. You can raise the exception, and you can handle it. That's it. Consider the way that Java approaches this same situation: all errors derive from a single Exception class. You can extend that class, adding other characteristics about an exception that you want to keep track of (error stack, context-sensitive data, etc.). An object instantiated from an Exception class is like any other kind of object in Java. You certainly can pass it as an argument to a method.

So PL/SQL doesn't let you do that with its native exceptions. This fact should not stop you from implementing your own exception "object." You can do so with Oracle object types or with a relational table of error information.

Regardless of implementation path, the key insight here is to distinguish between an error definition (error code is –1403, name is "no data found," cause is "implicit cursor did not find at least one row") and a particular *instance* of that error (I tried to select a company for this name and did not find any rows). There is, in other words, just one definition of the NO_DATA_FOUND exception, but there can be many different instances or occurrences of that exception. Oracle does not distinguish between these two representations of an error, but we certainly should—and we need to.

Here is an example of a simple exception object hierarchy to demonstrate the point. First, the base object type for all exceptions:

```
/* File on web: exception.ot */
CREATE TYPE exception_t AS OBJECT (
   name VARCHAR2(100),
   code INTEGER,
   description VARCHAR2(4000),
   help_text VARCHAR2(4000),
   recommendation VARCHAR2(4000),
   error_stack CLOB,
   call_stack CLOB,
   created_on DATE,
   created_by VARCHAR2(100)
   )
   NOT FINAL;
/
```

Next, I extend the base exception type for dynamic SQL errors by adding the sql_string attribute. When handling errors for dynamic SQL, it is very important to grab the string that is causing the problem so it can be analyzed later:

```
CREATE TYPE dynsql_exception_t UNDER exception_t (
   sql_string CLOB )
   NOT FINAL;
/
```

Here is another subtype of exception_t, this time specific to a given application entity: the employee. An exception that is raised for an employee-related error will include the employee ID and the foreign key to the rule that was violated:

```
CREATE TYPE employee_exception_t UNDER exception_t (
   employee_id INTEGER,
   rule_id INTEGER );
/
```

The complete specification of an error object hierarchy will include methods on the exception supertype to display error information or write it to the repository. I leave it to you to complete the hierarchy defined in the *exception.ot* file.

If you do not want to work with object types, you can take the approach I developed for the Quest Error Manager: I define a table of error definitions (Q$ERROR) and another table of error instances (Q$ERROR_INSTANCE), which contains information about specific occurrences of an error. All the context-specific data for an error instance is stored in the Q$ERROR_CONTEXT table.

Here is an example of the kind of code you would write with the Quest Error Manager API:

```
WHEN DUP_VAL_ON_INDEX
THEN
   q$error_manager.register_error (
         error_name_in => 'DUPLICATE-VALUE'
        ,err_instance_id_out => l_err_instance_id
        );
   q$error_manager.add_context (
         err_instance_id_in => l_err_instance_id
        ,name_in => 'TABLE_NAME', value_in => 'EMPLOYEES'
        );
   q$error_manager.add_context (
         err_instance_id_in => l_err_instance_id
        ,name_in => 'KEY_VALUE', value_in => l_employee_id
        );
   q$error_manager.raise_error_instance (
         err_instance_id_in => l_err_instance_id);
END;
```

If the duplicate value error was caused by the unique name constraint, I obtain an error instance ID or handle for the "DUPLICATE-VALUE" error. (That's right. I use error *names* here, entirely sidestepping issues related to error numbers.) Then I add context information for this instance (the table name and the primary key value that caused the problem). Finally, I raise the error instance, causing this block to fail and propagating the exception upward.

Just as you can pass data from your application into the error repository through the API, you can also retrieve error information with the get_error_info procedure. Here is an example:

```
BEGIN
   run_my_application_code;
EXCEPTION
   WHEN OTHERS
   THEN
      DECLARE
         l_error   q$error_manager.error_info_rt;
      BEGIN
         q$error_manager.get_error_info (l_error);
         DBMS_OUTPUT.put_line ('');
         DBMS_OUTPUT.put_line ('Error in DEPT_SAL Procedure:');
         DBMS_OUTPUT.put_line ('Code = ' || l_error.code);
         DBMS_OUTPUT.put_line ('Name = ' || l_error.NAME);
         DBMS_OUTPUT.put_line ('Text = ' || l_error.text);
         DBMS_OUTPUT.put_line ('Error Stack = ' || l_error.error_stack);
      END;
END;
```

These are just two of a number of different approaches to overcoming the limitations of the EXCEPTION type in PL/SQL. The bottom line is that there is no reason to accept the default situation, which is that you can only associate a code and message with the occurrence of an error.

Create Standard Templates for Common Error Handling

You cannot pass an exception to a program, which makes it very difficult to share standard error-handling sections among different PL/SQL blocks. You may find yourself writing the same handler logic over and over again, particularly when working with specific areas of functionality, such as file I/O with UTL_FILE. In these situations, you should take the time to create templates or starting points for such handlers.

Let's take a closer look at UTL_FILE (described further in Chapter 22). Prior to Oracle9*i* Database Release 2, UTL_FILE defined a number of exceptions in its package specification. However, Oracle neglected to provide error numbers for those exceptions via the EXCEPTION_INIT pragma. Consequently, if you did not handle a UTL_FILE exception by name, it would be impossible via SQLCODE to figure out what had gone wrong. Given this situation, you would probably want to set up a template for UTL_FILE programs that looked in part like this:

```
/* File on web: utlflexc.sql */
DECLARE
   l_file_id   UTL_FILE.file_type;

   PROCEDURE cleanup (file_in IN OUT UTL_FILE.file_type
                     ,err_in IN VARCHAR2 := NULL)
   IS
   BEGIN
      UTL_FILE.fclose (file_in);

      IF err_in IS NOT NULL
      THEN
         DBMS_OUTPUT.put_line ('UTL_FILE error encountered:');
         DBMS_OUTPUT.put_line (err_in);
      END IF;
   END cleanup;
BEGIN
   -- Body of program here

   -- Then clean up before exiting...
   cleanup (l_file_id);
EXCEPTION
   WHEN UTL_FILE.invalid_path
   THEN
      cleanup (l_file_id, 'invalid_path');
      RAISE;
   WHEN UTL_FILE.invalid_mode
   THEN
      cleanup (l_file_id, 'invalid_mode');
      RAISE;
END;
```

The key elements of this template include:

- A reusable cleanup program that ensures that the current file is closed before losing the handle to the file.
- The translation of the named exception to a string that can be logged or displayed so that you know precisely which error was raised.

Starting with Oracle9*i* Database Release 2, UTL_FILE does assign error codes to each of its exceptions, but you still need to make sure that files are closed when an error occurs and report on the error as consistently as possible.

Let's take a look at another UTL_FILE-related need for a template. Oracle9*i* Database Release 2 introduced the FREMOVE program to delete a file. UTL_FILE offers the DELETE_FAILED exception, raised when FREMOVE is unable to remove the file. After trying out this program, I discovered that FREMOVE may, in fact, raise any of several exceptions, including:

UTL_FILE.INVALID_OPERATION
: The file you asked UTL_FILE to remove does not exist.

UTL_FILE.DELETE_FAILED
: You (or the Oracle process) do not have the necessary privileges to remove the file, or the attempt failed for some other reason.

Thus, whenever you work with UTL_FILE.FREMOVE, you should include an exception section that distinguishes between these two errors, as in:

```
BEGIN
   UTL_FILE.fremove (dir, filename);
EXCEPTION
   WHEN UTL_FILE.delete_failed
   THEN
      DBMS_OUTPUT.put_line (
         'Error attempting to remove: ' || filename || ' from ' || dir);
      -- Then take appropriate action...

   WHEN UTL_FILE.invalid_operation
   THEN
      DBMS_OUTPUT.put_line (
         'Unable to find and remove: ' || filename || ' from ' || dir);
      -- Then take appropriate action...
END;
```

The *fileIO.pkg* available on the book's website offers a more complete implementation of such a template, in the context of an encapsulation of UTL_FILE.FREMOVE.

Making the Most of PL/SQL Error Management

It will be very difficult to create applications that are easy to use and debug unless you take a consistent, high-quality approach to dealing with errors.

Oracle PL/SQL's error management capabilities allow you to define, raise, and handle errors in very flexible ways. Limitations in its approach, however, mean that you will usually want to supplement the built-in features with your own application-specific code and tables.

I suggest that you meet this challenge by taking the following steps:

1. Study and understand how error raising and handling work in PL/SQL. It is not all completely intuitive. A prime example: an exception raised in the declaration section will *not* be handled by the exception section of that block.
2. Decide on the overall error management approach you will take in your application. Where and when do you handle errors? What information do you need to save, and how will you do that? How are exceptions propagated to the host environment? How will you handle deliberate, unfortunate, and unexpected errors?
3. Build a standard framework to be used by all developers; that framework will include underlying tables, packages, and perhaps object types, along with a well-defined process for using these elements. Don't resign yourself to PL/SQL's limitations. Work around them by enhancing the error management model.
4. Create templates that everyone on your team can use, making it easier to follow the standard than to write one's own error-handling code.