# 7 Handling Exceptions

PL/SQL has compile-time warnings and runtime errors. The latter are called exceptions.

## ☑ Define PL/SQL exceptions

Exception handling overview: 11.2

### Overview

- exceptions (plsql runtime errors) can arise from design faults, coding mistakes, hardware failures, and many other sources
- you cannot anticipate all possible exceptions, but you can write exception handlers that let your program continue to operate in their presence
- any plsql block can have an exception handling part, which can have one or more exception handlers, e.g.

```
EXCEPTION
  WHEN exname_1 THEN statements_1
  WHEN exname_2 OR exname_3 THEN statements_2
  WHEN OTHERS THEN statements_3
END;
```

- in the preceding syntax example, exname_n is the name of an exception and statements_n is one or more statements
- when an exception is raised in the executable part of the block, the executable part stops and control transfers to the exception-handling part. If exname_1 was raised, then statements_1 run. If either exname_2 or exname_3 was raised, then statements_2 run. If any other exception was raised, then statements_3 run
- after an exception handler runs, control transfers to the next statement of the enclosing block. If there is no enclosing block, then:
    - if the exception handler is in a subprogram, then control returns to the invoker, at the statement after the invocation
    - if the exception handler is in an anonymous block, then control transfers to the host environment (e.g. sqlplus)
- if an exception is raised in a block that has no exception handler for it, then the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a block has a handler for it or there is no enclosing block
- if there is no handler for the exception, then plsql returns an unhandled exception error to the invoker or host environment, which determines the outcome

- exception handler syntax:

```
WHEN {exception [OR ...] | OTHERS} THEN statement
```

- `OTHERS`: specifies all exceptions not explicitly specified in the exception-handling part of the block. If plsql raises such an exception, then the associated statements run.

> Note: oracle recommends that the last statement in the OTHERS exception handler be either RAISE or an invocation of the RAISE_APPLICATION_ERROR procedure. If you do not follow this practice, and PL/SQL warnings are enabled, you get PLW-06009

- In the exception-handling part of a block, the `WHEN OTHERS` exception handler is optional. It can appear only once, as the last exception handler in the exception-handling part of the block.

## Exception categories

- **Internally defined**
  - the runtime system raises internally defined exceptions implicitly (automatically). Examples of internally defined exceptions are `ORA-00060` (deadlock detected while waiting for resource) and `ORA-27102` (out of memory)
  - an internally defined exception always has an error code, but does not have a name unless plsql gives it one or you give it one
- **Predefined**
  - a predefined exception is an internally defined exception that plsql has given a name. For example, `ORA-06500` (PL/SQL: storage error) has the predefined name `STORAGE_ERROR`
- **User-defined**
  - you can declare your own exceptions in the declarative part of any plsql anonymous block, subprogram, or package
  - you must raise user-defined exceptions explicitly
- for a named exception, you can write a specific exception handler, instead of handling it with an `OTHERS` exception handler. A specific exception handler is more efficient than an `OTHERS` exception handler, because the latter must invoke a function to determine which exception it is handling (`SQLCODE`, `SQLERRM`)

## Advantages of exception handlers

- using exception handlers for error-handling makes programs easier to write and understand, and reduces the likelihood of unhandled exceptions
- without exception handlers, you must check for every possible error, everywhere that it might occur, and then handle it. It is easy to overlook a possible error or a place where it might occur, especially if the error is not immediately detectable. Error-handling code is scattered throughout the program
- with exception handlers, you need not know every possible error or everywhere that it might occur. You need only include an exception-handling part in each block where errors might occur. Error-handling code is isolated in the exception-handling parts of the blocks

- in the exception-handling part, you can include exception handlers for both specific and unknown errors. If an error occurs anywhere in the block (including inside a sub-block), then an exception handler handles it
- if multiple statements use the same exception handler, and you want to know which statement failed, you can use locator variables, e.g.

```
BEGIN
  locator := 1;
  statements1;
  locator := 2;
  statements2;
EXCEPTION
  WHEN ... THEN DBMS_OUTPUT.PUT_LINE('error in statements' || locator)
```

- you determine the precision of your error-handling code. You can have a single exception handler for all division-by-zero errors, bad array indexes, and so on. You can also check for errors in a single statement by putting that statement inside a block with its own exception handler

## Guidelines for avoiding and handling exceptions

- use both error-checking code and exception handlers
  - use error-checking code wherever bad input data can cause an error
  - examples of bad input data are incorrect or null actual parameters and queries that return no rows or more rows than you expect
  - test your code with different combinations of bad input data to see what potential errors arise
  - sometimes you can use error-checking code to avoid raising an exception, e.g using if or case statements, using cursor attributes with if, etc.
- add exception handlers wherever errors can occur
  - errors are especially likely during arithmetic calculations, string manipulation, and database operations
  - errors can also arise from problems that are independent of your code—for example, disk storage or memory hardware failure—but your code still must take corrective action
- design your programs to work when the database is not in the state you expect
  - for example, a table you query might have columns added or deleted, or their types might have changed
  - you can avoid problems by declaring scalar variables with `%TYPE` qualifiers and record variables to hold query results with `%ROWTYPE` qualifiers
- whenever possible, write exception handlers for named exceptions instead of using `OTHERS` exception handlers
  - learn the names and causes of the predefined exceptions
  - if you know that your database operations might raise specific internally defined exceptions that do not have names, then give them names so that you can write exception handlers specifically for them
- have your exception handlers output debugging information

    - if you store the debugging information in a separate table, do it with an autonomous routine, so that you can commit your debugging information even if you roll back the work that the main subprogram did
- for each exception handler, carefully decide whether to have it commit the transaction, roll it back, or let it continue
    - regardless of the severity of the error, you want to leave the database in a consistent state and avoid storing bad data
- avoid unhandled exceptions by including an `OTHERS` exception handler at the top level of every plsql program
    - make the last statement in the `OTHERS` exception handler either `RAISE` or an invocation of the `RAISE_APPLICATION_ERROR` procedure. (If you do not follow this practice, and plsql warnings are enabled, then you get `PLW-06009`

## ☑ Recognize unhandled exceptions

Unhandled exceptions: 11.9

- if there is no handler for a raised exception, plsql returns an unhandled exception error to the invoker or host environment, which determines the outcome

- if a **stored subprogram** exits with an unhandled exception, plsql does not roll back database changes made by the subprogram Careful with this one, if you try it out you will see that a rollback still does happen, and think this is wrong, but it is in fact correct (Oracle Community explanation):

```
CREATE TABLE dep (id number);

CREATE OR REPLACE PROCEDURE countrows(tab_name varchar2, block_name
varchar2) AS
  n number;
BEGIN
  EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || tab_name INTO n;
  DBMS_OUTPUT.PUT_LINE(n||' rows in '||tab_name||' in block '||block_name);
END countrows;
/
CREATE OR REPLACE PROCEDURE depfill AS
BEGIN
  INSERT INTO dep VALUES (1);
  countrows('dep', 'depfill');
  RAISE ZERO_DIVIDE;
END depfill;
/
BEGIN
  depfill;
EXCEPTION
  WHEN OTHERS THEN
    countrows('dep', 'anon block');
    RAISE;
END;
/
```

```
-- output
1 rows in dep in block depfill
1 rows in dep in block anon block
Error ...

exec countrows('dep', 'after error');
0 rows in dep in block after error
```

the stored subprogram does in fact not roll back the changes after it exits with unhandled exception, but the exception is then propagated to an anonymous block, which always does a rollback when exception unhandled

> Note: if commit was specified before error statement, data is committed even if unhandled exception

- the `FORALL` statement runs one DML statement multiple times, with different values in the `VALUES` and `WHERE` clauses. If one set of values raises an unhandled exception, then plsql rolls back all database changes made earlier in the `FORALL` statement.
- > Tip: avoid unhandled exceptions by including an `OTHERS` exception handler at the top level of every plsql program

## ☑Handle different types of exceptions (internally defined exceptions, predefined exceptions and user-defined exceptions)

Internally defined: 11.3

Predefined: 11.4

User-defined: 11.5

Redeclared predefined: 11.6

Raising exceptions: 11.7

Error codes and messages: 11.10

After exception handling: 11.11 - 11.12

### Internally defined exception

- **internally defined exceptions** (ORA-n errors) are described in Oracle Database Error Messages Reference. The runtime system raises them implicitly (automatically)
- an internally defined exception does not have a name unless either plsql gives it one or you give it one
- if you know that your database operations might raise specific internally defined exceptions that do not have names, then give them names so that you can write exception handlers specifically for them. Otherwise, you can handle them only with `OTHERS` exception handlers
- to give a name to an internally defined exception, do the following in the declarative part of the appropriate anonymous block, subprogram, or package:
    1. declare the name. An exception name declaration has this syntax: `exception_name EXCEPTION;`

2. associate the name with the error code of the internally defined exception. The syntax is: `PRAGMA EXCEPTION_INIT (exception_name, error_code)`

- > Note: an internally defined exception with a user-declared name is still an internally defined exception, not a user-defined exception\

- `EXCEPTION_INIT` Pragma
  - the `EXCEPTION_INIT` pragma associates a user-defined exception name with an error code
  - the `EXCEPTION_INIT` pragma can appear only in the same declarative part as its associated exception, anywhere after the exception declaration
  - syntax: `PRAGMA EXCEPTION_INIT (exception, error_code) ;`
  - exception is the name of a previously declared user-defined exception
  - error_code can be either 100 (the numeric code for "no data found" that "SQLCODE Function" returns) or any negative integer greater than -10000000 except -1403 (another numeric code for "no data found")
  - If two `EXCEPTION_INIT` pragmas assign different error codes to the same user-defined exception, then the later pragma overrides the earlier pragma

### Predefined exceptions

- **predefined exceptions** are internally defined exceptions that have predefined names, which plsql declares globally in the package `STANDARD`
- the runtime system raises predefined exceptions implicitly (automatically)
- because predefined exceptions have names, you can write exception handlers specifically for them
- names and error codes of the predefined exceptions

### User-defined exceptions

- you can declare your own exceptions in the declarative part of any PL/SQL anonymous block, subprogram, or package
- an exception name declaration has this syntax: `exception_name EXCEPTION;`
- you must raise a user-defined exception explicitly

### Redeclared predefined exceptions

- Oracle recommends against redeclaring predefined exceptions—that is, declaring a user-defined exception name that is a predefined exception name
- If you redeclare a predefined exception, your local declaration overrides the global declaration in package `STANDARD`. Exception handlers written for the globally declared exception become unable to handle it—unless you qualify its name with the package name `STANDARD`

### Raising exceptions explicitly

- To raise an exception explicitly, use either the `RAISE` statement or `RAISE_APPLICATION_ERROR` procedure
- The `RAISE` statement explicitly raises an exception
  - Outside an exception handler, you must specify the exception name, e.g. `RAISE my_exception;`, `RAISE ZERO_DIVIDE;` etc.

    - Inside an exception handler, if you omit the exception name, the `RAISE` statement reraises the current exception
    - **Raising Internally Defined Exception with RAISE Statement**
        - Although the runtime system raises internally defined exceptions implicitly, you can raise them explicitly with the RAISE statement if they have names
        - An exception handler for a named internally defined exception handles that exception whether it is raised implicitly or explicitly
    - **Reraising Current Exception with RAISE Statement**
        - In an exception handler, you can use the RAISE statement to"reraise" the exception being handled. Reraising the exception passes it to the enclosing block, which can handle it further. If the enclosing block cannot handle the reraised exception, then the exception propagates
        - When reraising the current exception, you need not specify an exception name
- `RAISE_APPLICATION_ERROR` procedure
    - You can invoke the `RAISE_APPLICATION_ERROR` procedure (defined in the `DBMS_STANDARD` package) only from a stored subprogram or method
    - Typically, you invoke this procedure to raise a user-defined exception and return its error code and error message to the invoker
    - syntax: `RAISE_APPLICATION_ERROR (error_code, message[, {TRUE | FALSE}]);`
        - you must have assigned error_code to the user-defined exception with the `EXCEPTION_INIT` pragma
        - the error_code is an integer in the range -20000..-20999
        - the message is a character string of at most 2048 bytes
        - If you specify `TRUE`, PL/SQL puts error_code on top of the error stack. Otherwise, PL/SQL replaces the error stack with error_code

## Retrieving error code and error message

In an exception handler, for the exception being handled:

- You can retrieve the error code with the PL/SQL function `SQLCODE`
    - In an exception handler, the `SQLCODE` function returns the numeric code of the exception being handled. (Outside an exception handler, `SQLCODE` returns 0)
    - For an internally defined exception, the numeric code is the number of the associated Oracle Database error. This number is negative except for the error "no data found", whose numeric code is +100
    - For a user-defined exception, the numeric code is either +1 (default) or the error code associated with the exception by the `EXCEPTION_INIT` pragma
    - If a function invokes `SQLCODE`, and you use the `RESTRICT_REFERENCES` pragma to assert the purity of the function, then you cannot specify the constraints `WNPS` and `RNPS`
- You can retrieve the error message with either:
    - The PL/SQL function `SQLERRM`
        - The `SQLERRM` function returns the error message associated with an error code
        - If a function invokes `SQLERRM`, and you use the `RESTRICT_REFERENCES` pragma to assert the purity of the function, then you cannot specify the constraints `WNPS` and `RNPS`
        - > Note: `DBMS_UTILITY.FORMAT_ERROR_STACK` is recommended over `SQLERRM`, unless you use the `FORALL` statement with its `SAVE EXCEPTIONS` clause

        - syntax: `SQLERRM [ (error_code) ]`
        - Like `SQLCODE`, `SQLERRM` without error_code is useful only in an exception handler. Outside an exception handler, or if the value of error_code is zero, `SQLERRM` returns `ORA-0000`
        - If the value of error_code is +100, `SQLERRM` returns `ORA-01403`
        - If the value of error_code is a positive number other than +100, `SQLERRM` returns this message: `-error_code: non-ORACLE exception`
        - If the value of error_code is a negative number whose absolute value is an Oracle Database error code, `SQLERRM` returns the error message associated with that error code
        - If the value of error_code is a negative number whose absolute value is not an Oracle Database error code, `SQLERRM` returns this message: `ORA-error_code: Message error_code not found; product=RDBMS; facility=ORA`
        - This function returns a maximum of 512 bytes, which is the maximum length of an Oracle Database error message (including the error code, nested messages, and message inserts such as table and column names)
    - The package function `DBMS_UTILITY.FORMAT_ERROR_STACK`
        - This function formats the current error stack. This can be used in exception handlers to look at the full error stack
        - This function returns the full error stack, up to 2000 bytes
        - Oracle recommends using `DBMS_UTILITY.FORMAT_ERROR_STACK`, except when using the `FORALL` statement with its `SAVE EXCEPTIONS` clause
- A SQL statement cannot invoke `SQLCODE` or `SQLERRM`. To use their values in a SQL statement, assign them to local variables first
- Other error message and code utilities:
    - `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE`
        - This function displays the call stack at the point where an exception was raised, even if the subprogram is called from an exception handler in an outer scope
        - The output is similar to the output of the `SQLERRM` function, but not subject to the same size limitation
        - A `NULL` string is returned if no error is currently being handled
    - `UTL_CALL_STACK` package
        - subprograms in this package provide information about currently executing subprograms, including subprogram names

## Continuing execution after handling exceptions

- After an exception handler runs, control transfers to the next statement of the enclosing block (or to the invoker or host environment if there is no enclosing block). The exception handler cannot transfer control back to its own block

- If you want to run the next statement after the error statement or statement to be handled , put the error statement in a subblock with its own exception handler, control will then transfer to that next statement after the exception in the subbock was handled, e.g.

```
DECLARE n number;
BEGIN
  SELECT 1/0 INTO n FROM DUAL; -- error statement, zero divide
  INSERT INTO emp (id) VALUES (1); -- statement does not execute, because of
```

```
previous error statement
EXCEPTION WHEN ZERO_DIVIDE THEN ... ;
END;
/
DECLARE n number;
BEGIN
  BEGIN
    SELECT 1/0 INTO n FROM DUAL; -- error statement, zero divide, inside
subblock
  EXCEPTION WHEN ZERO_DIVIDE THEN ... ; -- exception handled inside subblock
  END; -- control transfers to enclosing block, statement below

  INSERT INTO emp (id) VALUES (1); -- statement executes
EXCEPTION WHEN ZERO_DIVIDE THEN ... ;
END;
```

### Retrying transactions after handling exceptions

To retry a transaction after handling an exception that it raised, use this technique:

1. Enclose the transaction in a sub-block that has an exception-handling part
2. In the sub-block, before the transaction starts, mark a savepoint
3. In the exception-handling part of the sub-block, put an exception handler that rolls back to the savepoint and then tries to correct the problem
4. Put the sub-block inside a LOOP statement
5. In the sub-block, after the COMMIT statement that ends the transaction, put an EXIT statement

- If the transaction succeeds, the COMMIT and EXIT statements execute
- If the transaction fails, control transfers to the exception-handling part of the sub-block, and after the exception handler runs, the loop repeats Example in documentation

## ✅Propagate exceptions

Exception propagation: 11.8

- If an exception is raised in a block that has no exception handler for it, then the exception **propagates**. That is, the exception reproduces itself in successive enclosing blocks until either a block has a handler for it or there is no enclosing block
- If there is no handler for the exception, then PL/SQL returns an unhandled exception error to the invoker or host environment, which determines the outcome
- A user-defined exception can propagate beyond its scope (that is, beyond the block that declares it), but its name does not exist beyond its scope. Therefore, beyond its scope, a user-defined exception can be handled only with an `OTHERS` exception handler
    - take the following example: the inner block declares an exception named past_due, for which it has no exception handler. When the inner block raises past_due, the exception propagates to the outer block, where the name past_due does not exist. The outer block handles the exception with an `OTHERS` exception handler
    - If the outer block does not handle the user-defined exception, then an error occurs

- > Note: Exceptions cannot propagate across remote subprogram invocations. Therefore, a PL/SQL block cannot handle an exception raised by a remote subprogram

**Propagation of exceptions raised in declarations**

- An exception raised in a declaration propagates immediately to the enclosing block (or to the invoker or host environment if there is no enclosing block)
- Therefore, the exception handler must be in an enclosing or invoking block, not in the same block as the declaration
- example

```
BEGIN
  DECLARE
    i number := 1/0; -- zero divide error in inner block declaration
  BEGIN
    null;
  EXCEPTION
    WHEN ZERO_DIVIDE THEN ... -- exception not handled here, in same block
  END;
EXCEPTION
  WHEN ZERO_DIVIDE THEN ... -- exception handled here, in enclosing block
END;
```

**Propagation of exceptions raised in exception handlers**

- An exception raised in an exception handler propagates immediately to the enclosing block (or to the invoker or host environment if there is no enclosing block)
- Therefore, the exception handler must be in an enclosing or invoking block