Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Error handling in PL/pgSQL blocks

Error names and codes

Choosing an error handler

Error handling overhead

Handling Errors in a Block

Error handling is performed if there is an `EXCEPTION` section

Changes are rolled back to the savepoint at the beginning of the block

- an implicit savepoint is set if the block contains an `EXCEPTION` section

If there is a handler that matches the error

- error handler commands are executed
- the block completes successfully

If there is no suitable handler

- the block completes with an error

If a run-time error occurs within a block, the program (block, function) is usually aborted, and the current transaction enters the failure mode: it cannot be committed and can only be rolled back.

But an error can be caught and processed. It can be done by extending the block with an additional `EXCEPTION` section, which lists error conditions and provides operators to handle each of them.

In general, `EXCEPTION` is similar to the try-catch construct available in some programming languages (except for transaction specifics, or course).

Before an error is processed, all changes are rolled back to the savepoint that is implicitly set at the beginning of each block containing the `EXCEPTION` section. That's why it is forbidden to use `COMMIT` and `ROLLBACK` commands in such blocks.

But although `SAVEPOINT` and `ROLLBACK TO SAVEPOINT` commands are not supported by PL/pgSQL, you can still implicitly use savepoints and rollbacks to savepoints both in functions and procedures.

https://postgrespro.com/docs/postgresql/12/plpgsql-control-structures#PLPGSQL-ERROR-TRAPPING

Handling Errors in a Block

Let's take a look at a simple example.

```
=> CREATE TABLE t(id integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO t(id) VALUES (1);
```

```
INSERT 0 1
```

If there are no errors, all operators in a block are executed as usual:

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'The SELECT INTO operator has completed';
END;
$$;
```

```
NOTICE:  The SELECT INTO operator has completed
DO
```

Now let's insert a "redundant" row to trigger an error.

```
=> INSERT INTO t(id) VALUES (2);
```

```
INSERT 0 1
```

If there is no EXCEPTION section in a block, the operator execution is interrupted, and the whole block is considered to be completed with an error:

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'The SELECT INTO operator has completed';
END;
$$;
```

```
ERROR:  query returned more than one row
HINT:  Make sure the query returns a single row, or use LIMIT 1.
CONTEXT:  PL/pgSQL function inline_code_block line 5 at SQL statement
```

To catch an error, a block must have an EXCEPTION section, which defines one or more error handlers.

This construct works similar to CASE: conditions are parsed from top to bottom, the first suitable code path is selected, and its operators are executed.

What will be displayed?

```
=> DO $$
DECLARE
    n integer;
BEGIN
    INSERT INTO t(id) VALUES (3);
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'The SELECT INTO operator has completed';
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'No data';
    WHEN too_many_rows THEN
        RAISE NOTICE 'Too much data';
        RAISE NOTICE 'Rows in a table: %', (SELECT count(*) FROM t);
END;
$$;
```

```
NOTICE:  Too much data
NOTICE:  Rows in a table: 2
DO
```

The executed handler corresponds to the too_many_rows error. Note: if a handler is executed, the table contains two rows because of a rollback to an implicit savepoint at the beginning of the block.

Note the following subtlety: if an error occurs in the DECLARE section or within the EXCEPTION section of the handler

itself, it will be impossible to catch it in this block.

```
=> DO $$
DECLARE
    n integer := 1 / 0; -- an error is not trapped here
BEGIN
    RAISE NOTICE 'Success';
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Division by zero';
END;
$$;
```

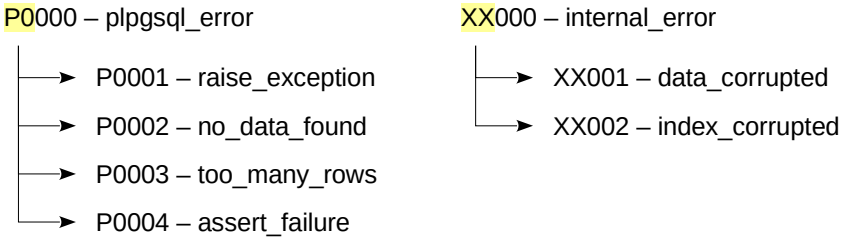
```
ERROR:  division by zero
CONTEXT:  SQL statement "SELECT 1 / 0"
PL/pgSQL function inline_code_block line 4 during statement block local variable initialization
```

Error Names and Codes

Information about an error

- error name
- five-character error code
- additional information: a short message, a detailed message, a hint, names of objects related to this error

A two-level hierarchy



5

Each possible error has a name and a code (a five-character string). `WHEN` clauses accept both error names and error codes.

All errors are classified into some sort of a two-level hierarchy. Each error class has a code that ends with three zeros; it corresponds to any error with the same first two characters in its code.

For example, the code 23000 defines the class that includes all errors dealing with violations of integrity constraints (such as 23502, which stands for not-null constraint violation, or 23505, which indicates unique constraint violation).

Thus, apart from regular errors, you can specify the whole error class by its name or code. Besides, you can use a special name *others* to trap any error (except for the fatal ones).

Apart from the name and code, each error can provide additional information to facilitate comprehension: a short error message, a detailed message, and a hint.

All errors are described in documentation in Appendix A:

https://postgrespro.com/docs/postgresql/12/errcodes-appendix

Errors can be not only trapped, but also raised programmatically.

https://postgrespro.com/docs/postgresql/12/plpgsql-errors-and-messages

Error Names and Codes

We have already seen error names; error codes are specified using SQLSTATE.

An error handler can return an error code and the corresponding message using the predefined variables SQLSTATE and SQLERRM.

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
EXCEPTION
    WHEN SQLSTATE 'P0003' OR no_data_found THEN -- there can be several conditions
        RAISE NOTICE '%: %', SQLSTATE, SQLERRM;
END;
$$;
```

```
NOTICE:  P0003: query returned more than one row
DO
```

Which error handler will be used?

```
=> DO $$
DECLARE
    n integer;
BEGIN
    SELECT id INTO STRICT n FROM t;
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'No data. %: %', SQLSTATE, SQLERRM;
    WHEN plpgsql_error THEN
        RAISE NOTICE 'Another error. %: %', SQLSTATE, SQLERRM;
    WHEN too_many_rows THEN
        RAISE NOTICE 'Too much data. %: %', SQLSTATE, SQLERRM;
END;
$$;
```

```
NOTICE:  Another error. P0003: query returned more than one row
DO
```

The first applicable handler is selected, plpgsql_error in this case. We will never get to the last error handler.

You can force an error using either its code or its name.

Here we use a special name “others,” which corresponds to any error that should be trapped (except for assertion failures and cases when the execution is aborted by user).

```
=> DO $$
BEGIN
    RAISE no_data_found;
EXCEPTION
    WHEN others THEN
        RAISE NOTICE '%: %', SQLSTATE, SQLERRM;
END;
$$;
```

```
NOTICE:  P0002: no_data_found
DO
```

If required, it is also possible to incorporate user-provided error codes that are not predefined, as well as pass some additional information (the example illustrates only some of the supported features):

```
=> DO $$
BEGIN
    RAISE SQLSTATE 'ERR01' USING
        message = 'A glitch in the Matrix.',
        detail  = 'The Matrix failure has occurred during execution',
        hint = 'Contact your system administrator';
END;
$$;
```

```
ERROR:  A glitch in the Matrix.
DETAIL:  The Matrix failure has occurred during execution
HINT:  Contact your system administrator
CONTEXT:  PL/pgSQL function inline_code_block line 3 at RAISE
```

Error handlers cannot get this information from variables; there is a special construct for analyzing such data in the code:

```
=> DO $$
DECLARE
    message text;
    detail text;
    hint text;
BEGIN
    RAISE SQLSTATE 'ERR01' USING
        message = 'Matrix failure',
        detail  = 'Irrecoverable matrix failure has occurred during execution',
        hint = 'Contact your system administrator';
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS
            message = message_text,
            detail = pg_exception_detail,
            hint = pg_exception_hint;
        RAISE NOTICE E'\nmessage = %\ndetail = %\nhint = %',
            message, detail, hint;
END;
$$;
```

```
NOTICE:
message = Matrix failure
detail = Irrecoverable matrix failure has occurred during execution
hint = Contact your system administrator
DO
```

Choosing a Handler

An unhandled error is sent one level up

- into the outer PL/pgSQL block, if available
- into the calling routine, if available

The search path of a handler is determined by the call stack

- i.e., it is not defined statically, it depends on the program execution

An unhandled error is passed to the client

- the transaction enters the failure mode and has to be rolled back by the client
- the error is registered in the server log

If none of the conditions listed in the `EXCEPTION` section is triggered, the error goes one level up.

If an error has occurred in the inner block of a nested structure, the server will search for a handler in the outer block. If there is no suitable handler either, the whole outer block will be treated as failed, while the error will be passed to the next nesting level, and so on.

If we have gone through the whole nested structure and have not found an appropriate error handler, the error goes further up to the level of the routine that has called this block. So you have to analyze the call stack to determine the order in which different error handlers will be tried.

If none of the available error handlers is triggered:

- the error message usually gets into the server log (the exact behavior depends on the server settings; see lecture “PL/pgSQL. Debugging”);

- the error is reported to the client that has initiated this operation in the database. The client has to face the fact: the transaction enters the failure mode, and it can only be rolled back.

It is up to the client to choose how to handle the error. For example, psql will display the error message and all the debugging information available. An end-user client may display a classic message like “contact your system administrator”.

Choosing a Handler

Let's take a look at several examples of choosing a handler in nested blocks. What will be displayed?

```
=> DO $$
BEGIN
    BEGIN
        SELECT 1/0;
        RAISE NOTICE 'The inner block has completed';
    EXCEPTION
        WHEN division_by_zero THEN
            RAISE NOTICE 'Error in the inner block';
    END;
    RAISE NOTICE 'The outer block has completed';
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Error in the outer block';
END;
$$;
```

```
NOTICE:  Error in the inner block
NOTICE:  The outer block has completed
DO
```

An error is handled in the same block where it has occurred. The outer block is executed as if there has been no error at all.

And now?

```
=> DO $$
BEGIN
    BEGIN
        SELECT 1/0;
        RAISE NOTICE 'The inner block has completed';
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'Error in the inner block';
    END;
    RAISE NOTICE 'The outer block has completed';
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Error in the outer block';
END;
$$;
```

```
NOTICE:  Error in the outer block
DO
```

The handler in the inner block is not applicable; the block completes with an error that is handled in the outer block.

Remember that the block containing an EXCEPTION section is rolled back to the implicit savepoint at the beginning of this block. In this case, all changes made in both blocks will be rolled back.

And now?

```
=> DO $$
BEGIN
    BEGIN
        SELECT 1/0;
        RAISE NOTICE 'The inner block has completed';
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'Error in the inner block';
    END;
    RAISE NOTICE 'The outer block has completed';
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'Error in the outer block';
END;
$$;
```

```
ERROR:  division by zero
CONTEXT:  SQL statement "SELECT 1/0"
PL/pgSQL function inline_code_block line 4 at SQL statement
```

None of the handlers is triggered, and the whole transaction is aborted.

There is usually no need to handle all possible errors in the server code. There is nothing wrong in passing an error to the client. In general, an error should be handled at the level where something meaningful can be done about it. So it makes sense to process an error within the database if it can be addressed on the server side (e.g., the operation can be repeated in case of a serialization failure). We'll talk about logging error messages in lecture "PL/pgSQL. Debugging."

Now let's take a look at an example that uses routines.

```
=> CREATE PROCEDURE foo()
AS $$
BEGIN
     CALL bar();
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CREATE PROCEDURE bar()
AS $$
BEGIN
    CALL baz();
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CREATE PROCEDURE baz()
AS $$
BEGIN
    PERFORM 1 / 0;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

What will happen if we call this procedure?

```
=> CALL foo();
```

```
ERROR:  division by zero
CONTEXT:  SQL statement "SELECT 1 / 0"
PL/pgSQL function baz() line 3 at PERFORM
SQL statement "CALL baz()"
PL/pgSQL function bar() line 3 at CALL
SQL statement "CALL bar()"
PL/pgSQL function foo() line 3 at CALL
```

The error message displays the call stack: top to bottom means inside out.

Note that this message (like many others) uses the term "function" instead of "procedure".

An error handler can also provide access to the call stack, but it will be presented as a single string:

```
=> CREATE OR REPLACE PROCEDURE bar()
AS $$
DECLARE
    msg text;
    ctx text;
BEGIN
    CALL baz();
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS
             msg = message_text,
             ctx = pg_exception_context;
        RAISE NOTICE E'\nError: %\nError stack:\n%\n', msg, ctx;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Let's check the result:

```
=> CALL foo();
```

```
NOTICE:
Error: division by zero
Error stack:
SQL statement "SELECT 1 / 0"
PL/pgSQL function baz() line 3 at PERFORM
SQL statement "CALL baz()"
PL/pgSQL function bar() line 6 at CALL
SQL statement "CALL bar()"
PL/pgSQL function foo() line 3 at CALL

CALL
```

Since a block with an EXCEPTION section creates an implicit savepoint, procedures cannot use COMMIT and ROLLBACK commands both in this block and in all the blocks up the call stack.

```
=> CREATE OR REPLACE PROCEDURE baz()
AS $$
BEGIN
    COMMIT;
END;
$$ LANGUAGE plpgsql;

CREATE PROCEDURE

=> CALL foo();

NOTICE:
Error: invalid transaction termination
Error stack:
PL/pgSQL function baz() line 3 at COMMIT
SQL statement "CALL baz()"
PL/pgSQL function bar() line 6 at CALL
SQL statement "CALL bar()"
PL/pgSQL function foo() line 3 at CALL

CALL
```

Overhead

Any block with the `EXCEPTION` section is executed slower

- because of setting an implicit savepoint

Additional costs are incurred in case of an error

- because of a rollback to the savepoint

Error handling can and should be used, but not overused

- anyway, PL/pgSQL is an interpreted language that uses SQL to compute expressions
- for most tasks, its speed is more than enough
- performance issues are usually related to queries, not to PL/pgSQL code

The mere inclusion of an `EXCEPTION` section already incurs overhead because it requires setting an implicit savepoint at the beginning of the block. If an error really occurs, the rollback to a savepoint increases the overhead even more.

So if there is a simple way to avoid exception handling, it's better to do without it. For example, you should not base your application logic on "exception juggling."

However, if error handing is really required, you should use it without doubt: errors can and must be handled regardless of the overhead. First, the PL/pgSQL language itself is quite slow because of interpreting instructions and constantly calling SQL to compute expressions. Second, its speed is usually still quite adequate. Yes, you can create a faster implementation in C, but what's the point? And third, the main performance issues are usually caused by bad query plans that affect query speed, not by the execution speed of procedural code (for details, see the QPT course that deals with query performance tuning).

But if there is an alternative that is both simpler and faster, it should certainly be preferred.

Overhead

To estimate the overhead, let's take a look at the following simple example.

Suppose we have a table with a text field that stores arbitrary data inserted by users (although usually a sign of bad design, it may sometimes be required). We need to extract all numbers into a separate column of a numeric type.

```
=> CREATE TABLE data(comment text, n integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO data(comment)
SELECT CASE
        WHEN random() < 0.01 THEN 'not a number' --  1%
        ELSE (random()*1000)::integer::text  -- 99%
    END
FROM generate_series(1,1000000);
```

```
INSERT 0 1000000
```

Let's solve this problem using error handling:

```
=> CREATE FUNCTION safe_to_integer_ex(s text) RETURNS integer
AS $$
BEGIN
    RETURN s::integer;
EXCEPTION
    WHEN invalid_text_representation THEN
        RETURN NULL;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Let's check the result:

```
=> \timing on
```

```
Timing is on.
```

```
=> UPDATE data SET n = safe_to_integer_ex(comment);
```

```
UPDATE 1000000
Time: 5733.175 ms (00:05.733)
```

```
=> \timing off
```

```
Timing is off.
```

```
=> SELECT count(*) FROM data WHERE n IS NOT NULL;
```

```
 count
--------
 989992
(1 row)
```

The following implementation of our function will check the format using a (slightly simplified) regular expression, without error handling:

```
=> CREATE FUNCTION safe_to_integer_re(s text) RETURNS integer
AS $$
BEGIN
    RETURN CASE
        WHEN s ~ '^\d+$' THEN s::integer
        ELSE NULL
    END;
END
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Let's check this implementation:

```
=> \timing on
```

```
Timing is on.
```

```
=> UPDATE data SET n = safe_to_integer_re(comment);
```

```
UPDATE 1000000
Time: 3233.919 ms (00:03.234)

=> \timing off

Timing is off.

=> SELECT count(*) FROM data WHERE n IS NOT NULL;

 count
--------
 989992
(1 row)
```

This implementation is almost two times faster. In this example, an exception has occurred in 1% of cases only. The more often it occurs, the more overhead will be incurred by rollbacks to the savepoint.

```
=> UPDATE data SET comment = 'not a number'; -- 100%

UPDATE 1000000

=> \timing on

Timing is on.

=> UPDATE data SET n = safe_to_integer_ex(comment);

UPDATE 1000000
Time: 9795.197 ms (00:09.795)

=> \timing off

Timing is off.
```

In some cases (which are not infrequent), you can do without error handling if you choose other suitable means.

Problem: return a row from a table or NULL, if there is no such row.

```
=> CREATE TABLE categories(code text UNIQUE, description text);

CREATE TABLE

=> INSERT INTO categories VALUES ('books','Books'), ('discs','CDs');

INSERT 0 2
```

A function with error handling:

```
=> CREATE FUNCTION get_cat_desc(code text) RETURNS text
AS $$
DECLARE
    desc text;
BEGIN
    SELECT c.description INTO STRICT desc
    FROM categories c
    WHERE c.code = get_cat_desc.code;

    RETURN desc;
EXCEPTION
    WHEN no_data_found THEN
        RETURN NULL;
END;
$$ STABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

Let's check that the function works as expected:

```
=> SELECT get_cat_desc('books');

 get_cat_desc
--------------
 Books
(1 row)


=> SELECT get_cat_desc('movies');

 get_cat_desc
--------------

(1 row)
```

Can we make it simpler?

Yes, we just need to remove STRICT or use a subquery:

```
=> CREATE OR REPLACE FUNCTION get_cat_desc(code text) RETURNS text
AS $$
BEGIN
    RETURN (SELECT c.description
            FROM categories c
            WHERE c.code = get_cat_desc.code);
END;
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

It's a good illustration that we do not need PL/pgSQL here at all: we can simply use SQL.

Let's check the result:

```
=> SELECT get_cat_desc('books');
```

```
 get_cat_desc
--------------
 Books
(1 row)
```

```
=> SELECT get_cat_desc('movies');
```

```
 get_cat_desc
--------------

(1 row)
```

Problem: update a table row with the specified ID; if there is no such row, insert it.

Here is the first approach. What is wrong with it?

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
DECLARE
    cnt integer;
BEGIN
    SELECT count(*) INTO cnt
    FROM categories c WHERE c.code = change.code;

    IF cnt = 0 THEN
        INSERT INTO categories VALUES (code, description);
    ELSE
        UPDATE categories c
        SET description = change.description
        WHERE c.code = change.code;
    END IF;
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Almost everything is bad here, starting from the fact that such a function will not work correctly at the Read Committed isolation level if there are several concurrent sessions. That's because the data in the database can change between the executed SELECT statement and the next operation.

It can be easily demonstrated by executing commands with a delay. For a change, let's see another implementation (also a bad one):

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
BEGIN
    UPDATE categories c
    SET description = change.description
    WHERE c.code = change.code;

    IF NOT FOUND THEN
        PERFORM pg_sleep(1); -- anything can happen here
        INSERT INTO categories VALUES (code, description);
    END IF;
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Now let's run this function in two different sessions, almost simultaneously:

```
=> SELECT change('games', 'Games');
```

```
  => SELECT change('games', 'Games');
```

```
  ERROR:  duplicate key value violates unique constraint "categories_code_key"
  DETAIL:  Key (code)=(games) already exists.
  CONTEXT:  SQL statement "INSERT INTO categories VALUES (code, description)"
  PL/pgSQL function change(text,text) line 9 at SQL statement
```

```
 change
--------

(1 row)
```

A correct solution can be implemented using error handling:

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
BEGIN
    LOOP
        UPDATE categories c
        SET description = change.description
        WHERE c.code = change.code;

        EXIT WHEN FOUND;
        PERFORM pg_sleep(1); -- for the demo

        BEGIN
            INSERT INTO categories VALUES (code, description);
            EXIT;
        EXCEPTION
            WHEN unique_violation THEN NULL;
        END;
    END LOOP;
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Let's check the result.

```
=> SELECT change('vynil', 'Vynil records');
```

```
  => SELECT change('vynil', 'Vynil records');
```

```
   change
  --------

  (1 row)
```

```
 change
--------

(1 row)
```

Yes, now everything is correct.

But there is an easier way: you can use a special flavor of the INSERT command that attempts to insert a row and performs an update if a conflict occurs. Again, all you need is pure SQL.

```
=> CREATE OR REPLACE FUNCTION change(code text, description text)
RETURNS void
AS $$
    INSERT INTO categories VALUES (code, description)
    ON CONFLICT(code)
        DO UPDATE SET description = change.description;
$$ VOLATILE LANGUAGE sql;
```

```
CREATE FUNCTION
```

Problem: make sure that the data is processed by one transaction at a time (at the Read Committed isolation level).

Using the same table, let's assume that the category sometimes requires a special single-threaded processing. The function can be declared as follows:

```
=> CREATE OR REPLACE FUNCTION process_cat(code text) RETURNS text
AS $$
BEGIN
    PERFORM c.code FROM categories c WHERE c.code = process_cat.code
        FOR UPDATE NOWAIT;
    PERFORM pg_sleep(1); -- the processing itself
    RETURN 'The category has been processed';
EXCEPTION
    WHEN lock_not_available THEN
        RETURN 'Another process is already processing this category';
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Let's check that everything is correct:

```
=> SELECT process_cat('books');
```

```
  => SELECT process_cat('books');

                      process_cat
  ------------------------------------------------------
   Another process is already processing this category
  (1 row)
```

```
          process_cat
--------------------------------
 The category has been processed
(1 row)
```

But this problem can also be solved without error handling if we use advisory locks:

```
=> CREATE OR REPLACE FUNCTION process_cat(code text) RETURNS text
AS $$
BEGIN
    IF pg_try_advisory_lock(hashtext(code)) THEN
        PERFORM pg_sleep(1); -- the processing itself
        RETURN 'The category has been processed';
    ELSE
        RETURN 'Another transaction is already processing this category';
    END IF;
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Let's check the result:

```
=> SELECT process_cat('books');
```

```
  => SELECT process_cat('books');

                         process_cat
  ------------------------------------------------------
   Another transaction is already processing this category
  (1 row)
```

```
          process_cat
--------------------------------
 The category has been processed
(1 row)
```

Let's see an example where we cannot do without error handling.

Problem: process a set of documents; a processing error of a particular document should not result in a general failure.

```
=> CREATE TYPE doc_status AS ENUM -- enumeration type
    ('READY', 'ERROR', 'PROCESSED');
```

```
CREATE TYPE
```

```
=> CREATE TABLE documents(
    id integer,
    version integer,
    status doc_status,
    message text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO documents(id, version, status)
SELECT id, 1, 'READY' FROM generate_series(1,100) id;
```

```
INSERT 0 100
```

A procedure that processes a single document can sometimes result in an error:

```
=> CREATE PROCEDURE process_one_doc(id integer)
AS $$
BEGIN
    UPDATE documents d
    SET version = version + 1
    WHERE d.id = process_one_doc.id;
    -- processing can take a while
    IF random() < 0.05 THEN
        RAISE EXCEPTION 'Catastrophic failure';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

Now let's create a procedure that processes all documents. It loops through the documents to process them one by one and catches an error if required.

Note that transactions are committed outside of the block that contains the EXCEPTION section.

```
=> CREATE PROCEDURE process_docs()
AS $$
DECLARE
    doc record;
BEGIN
    FOR doc IN (SELECT id FROM documents WHERE status = 'READY')
    LOOP
        BEGIN
            CALL process_one_doc(doc.id);

            UPDATE documents d
            SET status = 'PROCESSED'
            WHERE d.id = doc.id;
        EXCEPTION
            WHEN others THEN
                UPDATE documents d
                SET status = 'ERROR', message = sqlerrm
                WHERE d.id = doc.id;
        END;
        COMMIT; -- there is a separate transaction for each document
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

You can set up a similar processing using a function, but then all documents will be handled within a single common transaction, which can be a problem if processing takes a long time. This question is discussed at length in the DEV2 course.

Let's check the result:

```
=> CALL process_docs();
```

```
CALL
```

```
=> SELECT d.status, d.version, count(*)::integer
FROM documents d
GROUP BY d.status, d.version;
```

```
  status   | version | count
-----------+---------+-------
 ERROR     |       1 |     5
 PROCESSED |       2 |    95
(2 rows)
```

As you can see, some of the documents have not been processed, but it has not affected the rest.

It is convenient that the information about the occurred errors is stored in the table itself:

```
=> SELECT * FROM documents d WHERE d.status = 'ERROR';
```

```
 id | version | status |       message
----+---------+--------+----------------------
 18 |       1 | ERROR  | Catastrophic failure
 77 |       1 | ERROR  | Catastrophic failure
 80 |       1 | ERROR  | Catastrophic failure
 65 |       1 | ERROR  | Catastrophic failure
 85 |       1 | ERROR  | Catastrophic failure
(5 rows)
```

Please note once again that if an error occurs, the changes are rolled back to the savepoint at the beginning of the block; that's why documents with the ERROR status have not changed and still have version 1.

Summary

The search for an error handler is performed “inside out,” i.e., starting from the most inner block in the nested structure and going up the call stack

An implicit savepoint is set at the beginning of the block that contains `EXCEPTION`; if an error occurs, a rollback to this savepoint is performed

An unhandled error aborts the transaction; the error message is passed to the client and registered in the server log

Error handling incurs overhead

Practice

1. Specifying one and the same author several times when adding a book causes an error.
   Change the `add_book` function: trap the unique constraint violation error and force an error with a meaningful message instead.

   Test these changes in the application.

Task 1. To determine the name of the error that has to be trapped, catch all errors (`WHEN OTHERS`) and display the required information (by raising another error with the corresponding text).

Then remember to replace `WITH OTHERS` with a specific error: let all other error types be handled at a higher level if there is no opportunity to do anything useful in this particular place of the code.

(In real life, unique constraint violations should not be handled either: it is better to forbid entering the same author twice at the application level.)

Task 1. Processing Duplicated Author Names when Adding Books

```
=> CREATE OR REPLACE FUNCTION add_book(title text, authors integer[])
RETURNS integer
AS $$
DECLARE
    book_id integer;
    id integer;
    seq_num integer := 1;
BEGIN
    INSERT INTO books(title)
        VALUES(title)
        RETURNING books.book_id INTO book_id;
    FOREACH id IN ARRAY authors LOOP
        INSERT INTO authorship(book_id, author_id, seq_num)
            VALUES (book_id, id, seq_num);
        seq_num := seq_num + 1;
    END LOOP;
    RETURN book_id;
EXCEPTION
    WHEN unique_violation THEN
        RAISE EXCEPTION 'One and the same author cannot be specified twice';
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Practice

1. Some languages use the construct `try` … `catch` … `finally` …, where `try` corresponds to `BEGIN`, `catch` corresponds to `EXCEPTION`, and the operators located in the `finally` block are always triggered regardless of whether an exception has occurred and whether it has been processed by the `catch` block. Find a way to achieve a similar effect in PL/pgSQL.
2. Compare the call stacks returned by `GET STACKED DIAGNOSTICS` with `pg_exception_context` and `GET [CURRENT] DIAGNOSTICS` with `pg_context`.
3. Create the `getstack` function that returns the current call stack as a string array. The `getstack` function itself must not be a part of the stack.

Task 1. The easiest way to do it is to simply repeat `finally` operators in several places. But you should try to come up with a solution that avoids code duplication.

Task 2. Take a look at documentation first:

https://postgrespro.com/docs/postgresql/12/plpgsql-statements#PLPGSQL-STATEMENTS-DIAGNOSTICS

https://postgrespro.com/docs/postgresql/12/plpgsql-control-structures#PLPGSQL-EXCEPTION-DIAGNOSTICS

Task 1. Try-catch-finally

```
=> CREATE DATABASE plpgsql_exceptions;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_exceptions
```

```
You are now connected to database "plpgsql_exceptions" as user "student".
```

The problem is that “finally” operators must always be executed, even if there is an error in a “catch” operator (in the EXCEPTION block).

A possible solution is to use two nested blocks and a dummy exception that is called if the inner block has completed successfully. It allows grouping all “finally” operators in a single location, namely in the exception handler of the outer block.

```
=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'try operators';
        --
        RAISE NOTICE '...no exception';
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'catch operators';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'finally operators';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END;
$$;
```

```
NOTICE:  try operators
NOTICE:  ...no exception
NOTICE:  finally operators
DO
```

```
=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'try operators';
        --
        RAISE NOTICE '...a handled exception';
        RAISE no_data_found;
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'catch operators';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'finally operators';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END;
$$;
```

```
NOTICE:  try operators
NOTICE:  ...a handled exception
NOTICE:  catch operators
NOTICE:  finally operators
DO
```

```
=> DO $$
BEGIN
    BEGIN
        RAISE NOTICE 'try operators';
        --
        RAISE NOTICE '...an unhandled exception';
        RAISE division_by_zero;
    EXCEPTION
        WHEN no_data_found THEN
            RAISE NOTICE 'catch operators';
    END;
    RAISE SQLSTATE 'ALLOK';
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'finally operators';
        IF SQLSTATE != 'ALLOK' THEN
            RAISE;
        END IF;
END;
$$;
```

```
NOTICE:  try operators
NOTICE:  ...an unhandled exception
NOTICE:  finally operators
ERROR:  division_by_zero
CONTEXT:  PL/pgSQL function inline_code_block line 7 at RAISE
```

In the proposed solution, all changes performed in the block are always rolled back, so it cannot be used for commands that change the database state. You should also keep in mind the overhead incurred by exception handling: this implementation is provided for training purposes only.

Task 2. GET DIAGNOSTICS

```
=> DO $$
DECLARE
    ctx text;
BEGIN
    RAISE division_by_zero;                       -- line 5
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS ctx = pg_exception_context;
        RAISE NOTICE E'stacked =\n%', ctx;
        GET CURRENT DIAGNOSTICS ctx = pg_context; -- line 10
        RAISE NOTICE E'current =\n%', ctx;
END;
$$;
```

```
NOTICE:  stacked =
PL/pgSQL function inline_code_block line 5 at RAISE
NOTICE:  current =
PL/pgSQL function inline_code_block line 10 at GET DIAGNOSTICS
DO
```

GET STACKED DIAGNOSTICS shows the call stack that has led to an error.

GET [CURRENT] DIAGNOSTICS shows the current call stack.

Task 3. A Call Stack as an Array

The function to use:

```
=> CREATE FUNCTION getstack() RETURNS text[]
AS $$
DECLARE
    ctx text;
BEGIN
    GET DIAGNOSTICS ctx = pg_context;
    RETURN (regexp_split_to_array(ctx, E'\n'))[2:];
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

To check how it works, let's create several functions that call each other:

```
=> CREATE FUNCTION foo() RETURNS integer
AS $$
BEGIN
    RETURN bar();
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE FUNCTION bar() RETURNS integer
AS $$
BEGIN
    RETURN baz();
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE FUNCTION baz() RETURNS integer
AS $$
BEGIN
    RAISE NOTICE '%', getstack();
    RETURN 0;
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> SELECT foo();
```

```
NOTICE:  {"PL/pgSQL function baz() line 3 at RAISE","PL/pgSQL function bar() line 3 at RETURN","PL/pgSQL function foo() line 3 at RETURN"}
 foo
-----
   0
(1 row)
```