

Error Management and Logging in 4D Applications

By Karim Meghraoui Technical Support Engineer, 4D Morocco.

Technical Note 25-09

Table of Contents

Table of Contents	2
Abstract.....	3
Introduction	3
Traditional error handling using the "ON ERR CALL " command	3
The Three Scopes of the ON ERR CALL Command.....	4
1. Local scope (ek local)	4
2. Global scope (ek global)	4
3. Component scope (ek errors from components).....	4
Practical Case: Error When Writing to a Read-Only File	5
Case 1: Without ON ERR CALL	5
Case 2: With ON ERR CALL("ErrorWriteDoc").....	6
Execution Hierarchy of Error Handlers.....	7
Retrieving the Current Error Handler	7
Error stacks	8
Aborting an Error	11
Modern Error Handling in 4D	12
Throw an immediate error with a code and message	12
Throwing an error via an object containing properties throw(errorObj)	13
Deferred error handling	14
Introduction of the new Try() keyword and the new Try, Catch and End try keywords.....	14
Using Last errors with Try and Try Catch.....	14
Use case 1: Using the new Try(expression) keyword	14
Use case 2: Using the new Try, Catch and End try keywords.....	15
Conclusion	16

Abstract

In any complex application, various types of errors are bound to occur, from missing files and unstable network connections to unexpected bugs. These errors generally fall into several categories: syntax errors, which result from that violates the language's rules and are typically caught at compile time, runtime errors, which arise during execution due to invalid operations like division by zero, logical errors, which produce incorrect results despite the program running smoothly, network errors, caused by connectivity issues such as timeouts or unreachable servers, and resource errors, which occur when the system runs out of memory or file handles.

Modern 4D development tools offer robust solutions to detect, analyze, and manage these errors efficiently. By integrating such tools, developers can facilitate diagnostics, maintain application stability, and ensure a seamless user experience.

Introduction

By default, errors trigger a blocking dialog box that displays technical details such as the method, line, code, and message. This approach presents several challenges for users, including interruptions to their workflow, messages that are difficult to understand, and the lack of options for automatic recovery or error logging. This technical note outlines how to address these issues by handling errors through both traditional and modern approaches.

Traditional error handling using the "ON ERR CALL " command

The **ON ERR CALL** command is a standard 4D mechanism that allows errors to be handled either centrally or locally. It makes it possible to install a custom error routine (called a handler), which will be automatically invoked when an error occurs during code execution. A handler is installed using the following syntax:

```
ON ERR CALL("MethodName "; ek scope)
```

For example, to install a global error handler:

```
ON ERR CALL("myGlobalErrorHandler"; ek global)
```

And to disable it:

```
ON ERR CALL(""); // or ON ERR CALL("", ek global)
```

When the command is enabled, it not only allows errors to be intercepted but also makes it possible to customize the messages displayed, log the errors, or decide whether to continue or stop execution. This management applies at three scope levels: local, global, or component-specific.

The Three Scopes of the ON ERR CALL Command

1. Local scope (ek local)

The local scope makes it possible to define an error handler for a specific block of code or method. This approach provides precise and targeted control over errors that may occur in a particular part of the program. The local handler always takes precedence over other types of handlers, which means it will be executed first if an error is caught within its scope.

Its use is strongly recommended around sensitive operations such as file access, database manipulation, or network communications. Once the operation is complete, it is essential to restore the previous handler with the command **ON ERR CALL**(\$previousErrorHandler) to avoid disrupting the rest of the application.

2. Global scope (ek global)

The global scope allows developers to define an error handler that applies in all execution contexts where no local handler is present. This makes it an ideal solution for centralizing error management in a 4D application.

It is recommended to install this global handler in the database's **On Startup** method, which ensures that error handling is active from the very start of the environment. This type of handler improves the application's consistency and resilience by capturing all errors that are not already managed locally.

3. Component scope (ek errors from components)

To enhance error management related to components in 4D, a global handler can be installed in the host database using the **ON ERR CALL** command with the ek errors from components scope. This handler intercepts only errors not handled by the components themselves. It therefore acts as a complement, rather than a replacement, to the component's internal handler (if one exists).

This configuration is particularly useful for preventing the appearance of 4D's default error dialog boxes within the execution context of components. It ensures more predictable behavior, improves the robustness of the host database, and enables centralized management of component-related anomalies.

To activate this mechanism, a component-specific error handler must be installed using the following command:

```
ON ERR CALL("myComponentErrorHandler"; ek errors from components)
```

Practical Case: Error When Writing to a Read-Only File

Consider a method that attempts to write to a file named example.txt located in the database folder. This file is deliberately set to read-only, which makes any write operation impossible and triggers an “Access Denied” error.

The behavior of 4D code in this context will be observed both with and without an active error handler through the **ON ERR CALL** command.

```
var $error; $colJSON : Collection
var $fileContent : Text
var $created : Boolean
var $objFile : Object
var $previousErrorHandler : Text

$previousErrorHandler:=Method called on error

LOG EVENT(Into 4D diagnostic log; "Runtime error!"; Error message)
var $fileHandle : 4D.FileHandle:=Folder(Database folder).file("example.txt").open()

If ($fileHandle#Null)
ON ERR CALL("ErrorWriteDoc")
$previousErrorHandler:=Method called on error
$fileHandle.writeLine("hello")
End if
```

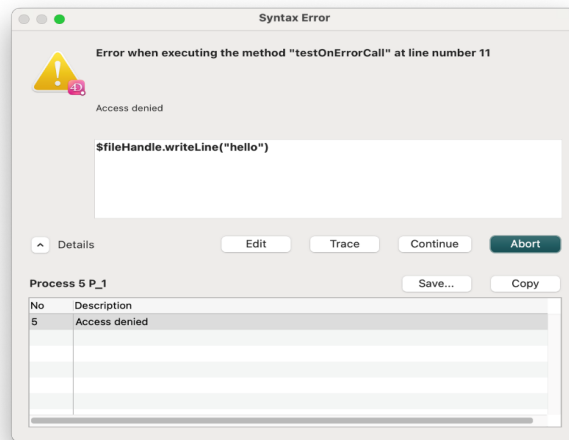
Case 1: Without ON ERR CALL

When the **ON ERR CALL** command is not used, any error encountered (such as an attempt to write to a read-only file) immediately triggers the appearance of a 4D system dialog box. The user must then intervene by clicking “Abort,” “Continue,” “Trace,” or “Edit.” Code execution is paused until this manual action is performed.

When calling the method:

```
$fileHandle.writeLine("hello")
```

4D triggers a system error. A dialog box appears, interrupting execution.



The user is forced to make a choice (Abort, Continue, etc.), making the process dependent on manual intervention.

Case 2: With ON ERR CALL("ErrorWriteDoc")

In 4D, the **ON ERR CALL("ErrorWriteDoc")** command allows a centralized error handler to be installed. When an error occurs, the ErrorWriteDoc method is automatically called to handle the incident without blocking code execution.

```
$error:=Last errors
If ($error.length=0)
$error:={}
End if
$error[$error.length-1].time:=String(Current date)+" "+String(Current time)
$error[$error.length-1].app:=Application info
$error[$error.length-1].version:=Application version
$error[$error.length-1].os:=System info.osVersion
$error[$error.length-1].method:=Error method
$error[$error.length-1].line:=Error line
$error[$error.length-1].formula:=Error formula
$obFile:=Folder(fk logs folder).file("errors.json")
If (Not($obFile.exists))
$created:=$obFile.create()
If (Not($created))
LOG EVENT(Into 4D diagnostic log; "Unable to create error file!"; Error message)
End if
$colJSON:=[]
Else
$fileContent:=$obFile.getText()
If ($fileContent#"")
$colJSON:=JSON Parse($fileContent)
Else
$colJSON:=[]
End if
End if
$colJSON.push($error)
```

```
$objFile.setText(JSON Stringify($colJSON; *))  
//ALERT("Write Error !")
```

This method begins by retrieving recent errors using Last errors. The most recent error is then enriched with valuable contextual information. Among these, Error method indicates the name of the method where the error occurred, Error line provides the relevant line number, and Error formula gives the line of code containing the error. These elements make it possible to pinpoint the exact location of the problem in the code.

Additional information is also added, such as the date, time, application version, and system version. All of this is then saved in an errors.json file, which serves as an error log. If the file already exists, its contents are updated; otherwise, it is created automatically.

Execution Hierarchy of Error Handlers

When 4D encounters an error, it follows a strict hierarchy to determine which handler to call:

1. If a local handler is defined (ek local), it is executed first.
2. Otherwise, if the error comes from a component, the component handler is called (ek errors from components).
3. Otherwise, the global handler (ek global) takes over.
4. If no handler is installed, 4D displays a system error dialog.

This hierarchy ensures a logical priority and allows different levels of error handling to be combined within the same application without conflict.

Retrieving the Current Error Handler

When using the **ON ERR CALL**("MethodName") command in 4D, a custom error handler is activated. This means that if an error occurs afterward, instead of 4D displaying a system error, the specified method is automatically called to handle the error.

However, if another error handler was already in place, it is important to save it before activating a new one. This allows it to be restored once the current process is complete, preventing disruption to the rest of the application.

For this purpose, the **Method called on error** command is used, which returns the name of the currently active error handler (or an empty string if none exists). It is stored in a variable, for example:

For this purpose, the **Method called on error** command is used, which returns the name of the currently active error handler (or an empty string if none is set). It is stored in a variable, for example:

```
$previousErrorHandler := Method called on error
```

Another handler can be temporarily activated with **ON ERR CALL**("MyMethod"). Sensitive operations, such as writing to a file, may then be performed, and the previous handler can finally be restored with:

```
ON ERR CALL($previousErrorHandler)
```

This line restores exactly the handler that was active initially. If none was defined beforehand, it automatically disables the handler (since the variable contains an empty string).

This ensures that the code remains reliable and does not interfere with other parts of the application that may implement separate error handling.

The **Method called on error** command accepts an optional parameter to retrieve the global error handler and the component error handler.

For local error handling:

```
$localHandler:=Method called on error
```

For global error handling:

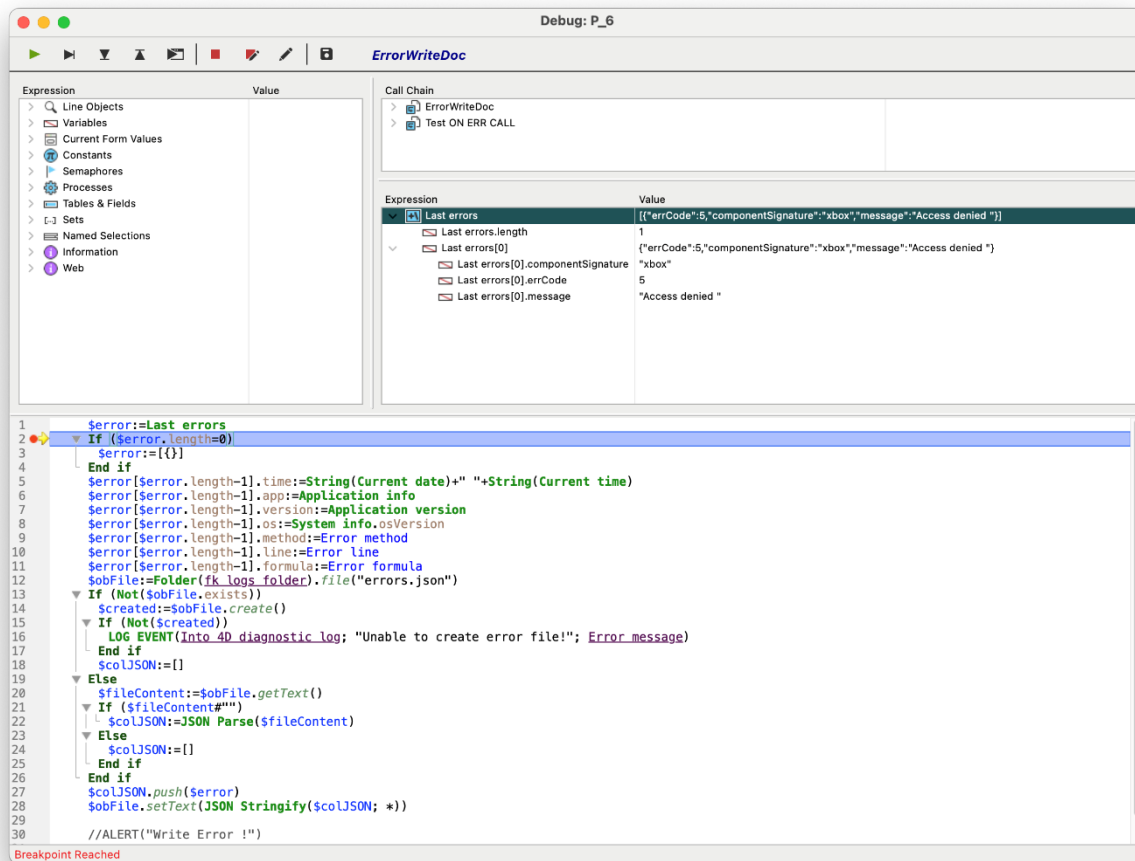
```
$globalHandler:=Method called on error(ek global)
```

For error handling in a component:

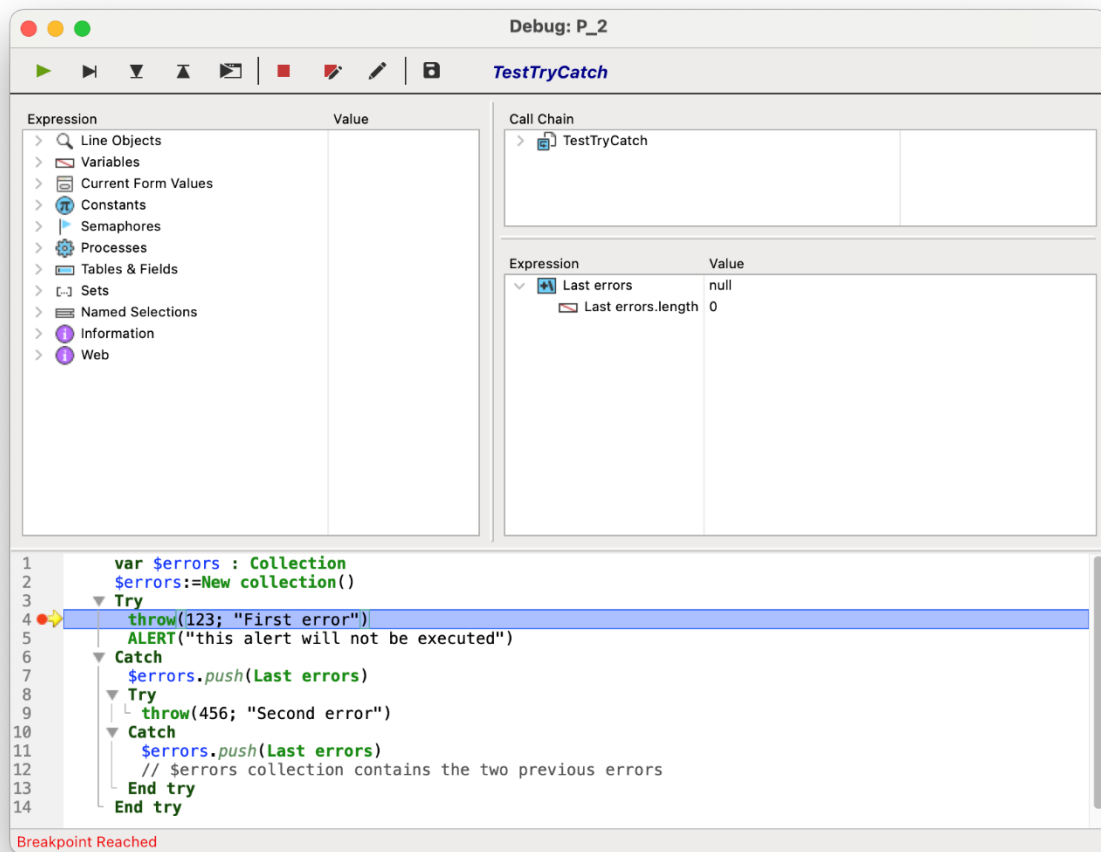
```
$componentHandler:=Method called on error(ek errors from components)
```

Error stacks

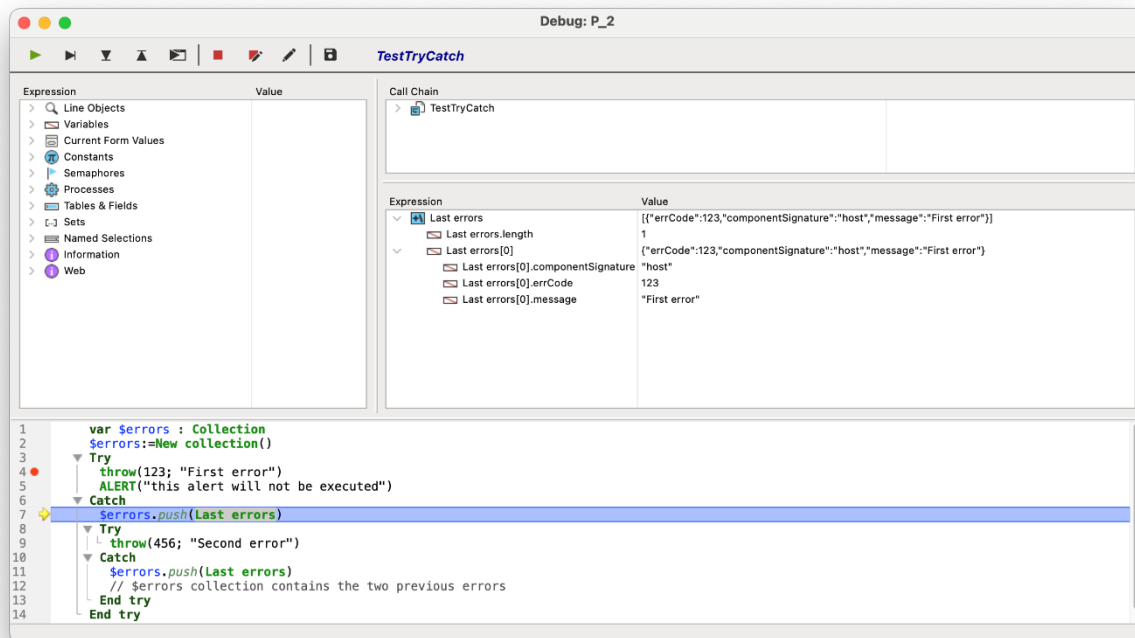
The **Last errors** command allows developers to retrieve the current error stack in the 4D application. It returns a collection of objects:



Each corresponding to an error, with information such as the error number, the associated message, and the signature of the internal component that caused the error. If no error has been recorded, the command returns null



The stack can also contain errors explicitly generated using the **throw** command.



This command must be used in a context suitable for error handling, such as a method installed via **ON ERR CALL** or inside a **Try** or **Try/Catch** block.

Aborting an Error

Regardless of which handler is executed, the **ABORT** command is always effective and allows code execution to be stopped.

The **ABORT** command is designed to be used within a project method for error handling defined via **ON ERR CALL**. When no such method is installed, 4D displays its default error dialog and interrupts execution according to the context (form, menu, process, etc.).

However, if an interception method is defined, 4D no longer displays its dialog box and continues execution after calling the error handler. Certain errors can then be managed programmatically. For critical or unrecoverable errors, it is recommended to call the **ABORT** command to properly stop execution, as 4D would do by default.

Note: Although some use the ABORT command in other contexts, its official use is reserved for project methods handling error interception.

Modern Error Handling in 4D

Since 4D version 20 R2, the **throw()** command allows developers to manually generate an error at a specific point in the code. It accepts an error code and/or a custom message, and behaves like a standard 4D error:

- If no handler is defined (**ON ERR CALL**), a dialog box is displayed.
- Otherwise, the error is captured by the specified method.

This functionality facilitates proactive detection, logging, and structured error handling.

Throw an immediate error with a code and message

The **throw(code; message)** command raises an error instantly. This error can:

- Be caught by an **ON ERR CALL** method.
- Display an error dialog if no handler is active.

This allows for clearly signaling exceptional cases in the code.

Let's take a simple example: division by zero. The following code illustrates a Euclidean division method that generates an error, then arbitrarily returns zero when the divisor is null.

```
#DECLARE($dividend : Real; $divisor : Real)->$result : Real
If ($divisor=0)
    $result:=0
    throw(-12345; "Division by zero!")
Else
    $result:=( $dividend/$divisor)
End if
```

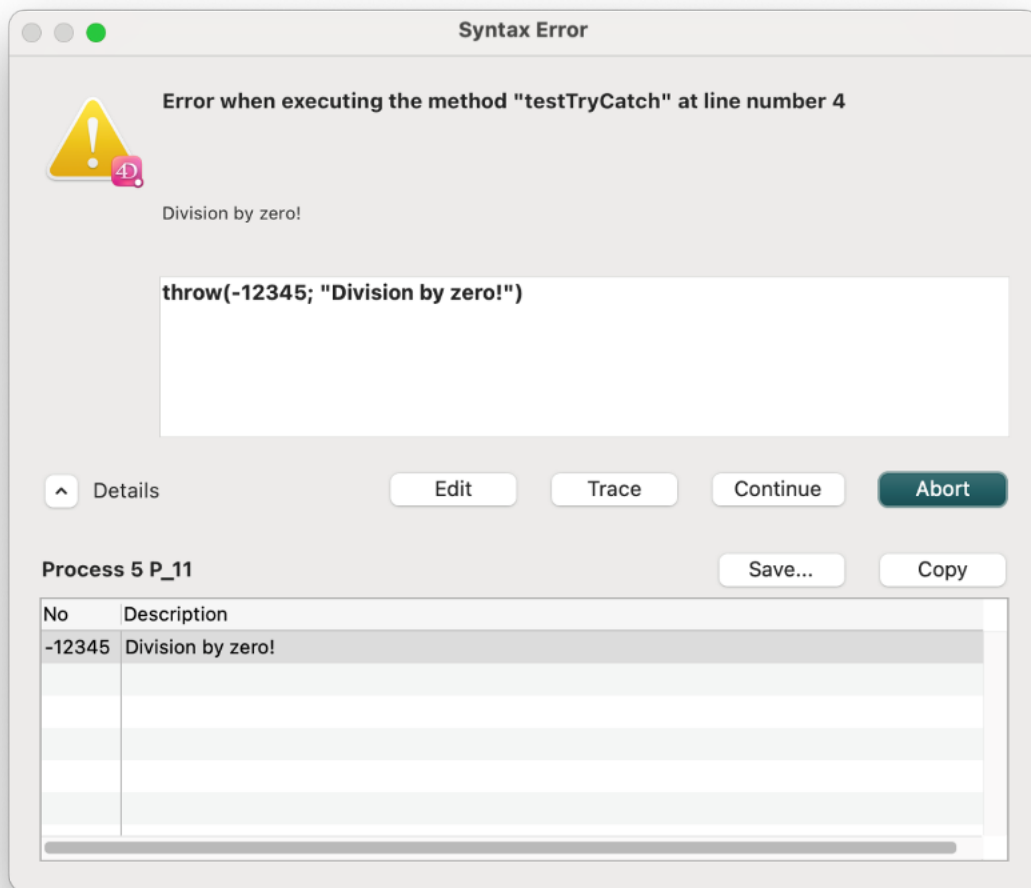
throw (-12345; "Division by zero !")

This instruction triggers an error with code -12345 and the message "Division by zero!".

If an interception method has been defined via **ON ERR CALL**, it is called to handle the error.

- Otherwise, 4D automatically displays the standard error dialog box, including the custom information : code -12345 and the message "Division by zero!" (Screenshot below).

The standard error display is the one shown just below.



Throwing an error via an object containing properties `throw(errorObj)`

The `errorObj` object allows for detailed and customized error handling. It can contain several properties:

- **componentSignature (text):** identifies the error source with a 4-letter signature. By default, "host" for the host database or "C001", "C002"... for components
- **errCode (numeric):** error code, -1 if not provided.
- **message (text):** error description, which can contain placeholders (e.g. {propertyName}). If empty or absent, a message is searched for in xliiff files with the name `ERR_{componentSignature}_{errCode}`.
- **deferred (boolean):** if true, the error is deferred until the end of the method or Try block (default: false).

It is possible to execute the command multiple times within the same method to generate multiple errors. By activating the deferred option, all these errors can be grouped and returned at once.

Deferred error handling

Deferred mode (deferred) allows postponing the triggering of an error until the end of a method or a Try block.

- In an application, if an **ON ERR CALL** handler is defined, it is automatically called at the end of the method.
- In a component, the deferred error stack can be transmitted to the host application, which then executes its own error handler.

The **Last errors** command allows retrieving this stack to analyze all errors that occurred in deferred mode.

Introduction of the new Try() keyword and the new Try, Catch and End try keywords

Since 4D versions 20 R4 and 20 R5, the 4D language offers a new way to handle runtime errors. Two major features are at the heart of this evolution:

- The **Try(expression)** instruction, introduced with 4D 20 R4, allows attempting an operation without blocking execution in case of an error.
- The **Try...Catch** block, introduced with 4D 20 R5, allows handling errors in a clear structure, by automatically capturing any exception thrown in the Try block.

These two tools rely on the **Last errors** command, which plays a central role in error recovery and analysis.

Using Last errors with Try and Try Catch

Last errors return the current error stack in 4D, as a collection of error objects, or Null if no error has occurred.

- The error stack includes objects sent by the throw command, if applicable.
- This command must be called from an error callback method installed by the **ON ERR CALL** command or in a **Try** or **Try/Catch** context.

Each error object in this collection contains useful information such as:

- **errorCode (type Number)**: Error code
- **message (type text)**: Error description
- **componentSignature (type text)**: Signature of the internal component that returned the error

Use case 1: Using the new Try(expression) keyword

In this example, the **Try(expression)** instruction is used to attempt to open and write to a text file named **example.txt**. This file has been set to read-only to intentionally cause an error. The code begins with a file opening attempt encapsulated in Try. If the opening succeeds, a FileHandle object is returned. Otherwise, Try returns Null and the error is automatically recorded in **Last errors**.

```
var $fileHandle : 4D.FileHandle:=Try(Folder(Database
folder).file("example.txt").open())
If ($fileHandle#Null)
    Try($fileHandle.writeLine("Hello"))
End if
If (Last errors#Null)
    ErrorWriteDoc
End if
```

Since the write operation fails due to read-only mode, the error is returned in **Last errors**. This command allows retrieving errors as a collection. If **Last errors** is not empty, a method is called to handle the error (The **ErrorWriteDoc** method is the one called in this example).

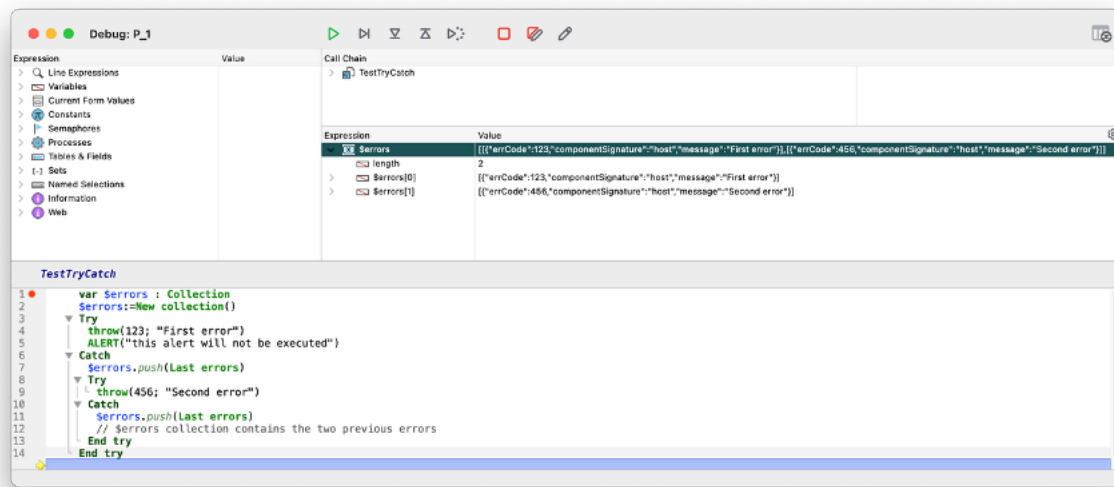
Use case 2: Using the new Try, Catch and End try keywords

In this second example, the **Try...Catch** blocks, available since 4D 20 R5, provide structured error handling. This structure allows wrapping multiple instructions with a Try block and immediately capturing any error with the Catch block. In the example, an error is intentionally generated with the **throw** command. As soon as this error is raised, the code in **Try** stops and the program goes directly to the Catch block. There, **Last errors** is retrieved and added to a collection. Then, a second **Try...Catch** block is used inside the first one to throw another error. In the end, the collection contains two errors. This shows how to handle multiple errors without stopping the program, and how to use **Last errors** to keep track of each one.

```
var $errors : Collection
$errors:=New collection()
Try
    throw(123; "First error")
    ALERT("this alert will not be executed")
Catch
    $errors.push(Last errors)
    Try
        throw(456; "Second error")
    Catch
        $errors.push(Last errors)
        // $errors collection contains the two previous errors
    End try
End try
```

At the end of this processing, the **\$errors** collection contains two error objects, each from a **Catch** block. These objects include information such as the **error code**, the **message defined by the developer**, and the **error name**. This not only allows

capturing errors without interrupting the program but also tracing them precisely and storing them for later analysis or logging.



Thus, whether using **Try(expression)** or **Try...Catch** blocks, the **Last errors** command remains the central element for identifying, retrieving and exploiting errors. In the first case, it must be queried manually, while in the second, it is used automatically in Catch blocks to extract captured errors. This modernization of error handling in 4D makes the code more readable, more robust, and above all more compliant with modern language standards.

Conclusion

Modern error handling in 4D provides a comprehensive and flexible toolkit for managing errors. Using **ON ERR CALL**, **throw()**, and **try & try/catch**, developers can control error flow precisely, isolate issues at the source, and maintain a clean and readable codebase. These mechanisms support detailed logging, enable consistent handling across local, global, and component contexts, and reduce reliance on manual intervention. By integrating these practices, applications become more robust, easier to debug, and maintainable, ensuring that 4D solutions can reliably scale in a mission-critical environment.