Contents

Yonder Academy – Progress training

1 Error handling

An error occurs when the ABL virtual machine (AVM)fails to successfully execute ABL code.

- Errors raised by the AVM are called **system errors**.
- Error raised programatically are called **application errors**.
- In some languages, unrecoverable errors are called exceptions. In ABL, the STOP condition represents unrecoverable errors.

Generally speaking, a condition differs from other run-time events in that it is **unexpected** and requires a **response** to restore application flow. In ABL, a condition always invokes a default response provides which is called **default error handling**. This robust default error handling protects persistent data while also providing branching options to restore application flow. You design your ABL application to accept default error handling, to suppress it, or to replace it with custom error handling.

2 ABL Block Essentials

ABL error handling starts at the statement level and is then handled by the nearest enclosing block. Default block properties and block statement options then determine how the block handles the error.

2.1 Basic blocks

- DO
- FOR
- REPEAT

These three blocks have a rich set of options that allow a larger set of basic block variations

- Basic blocks have few restrictions on where they can appear in ABL code.
- Flow of control (branching) with basic blocks is handled by references to block labels.
- Only basic blocks can iterate.
- Only basic blocks allow you to alter default error handling with the ON ERROR phrase.
- A DO block without options has no default block properties and is called a simple DO block. Therefore, a DO block does not have default error handling unless it is either a DO TRANSACTION block or a DO ON ERROR block.

2.2 End blocks

An end block is a block that defines end-of-block processing for the block that encloses it. End blocks are always part of another block and that block is called the associated block. End blocks must appear in the associated block after the last executable statement and before the END statement. The end blocks are:

- CATCH
- FINALLY

2.3 Routine-level blocks

A routine is a module of code that can be called or executed from another module by referencing a routine name. In ABL, a routine-level block can be "called" in a variety of ways. It can be:

- Executed by a RUN statement or a tool

- Triggered by a database event or a user-interface event
- Referenced in an assignment or expression
- Accessed from an object handle (system handle or class instance)

The routine-level blocks are:

- Procedure (also called an external procedure or .p file)
- Internal procedure
- User-defined function
- Database trigger procedure (.p file) and database trigger block (ON database-event statement)
- User-interface trigger
- Class method (user-defined method), constructor, destructor, and property accessor (GET and SET methods)

3 Conditions

ABL recognizes four conditions, all of which are keywords:

- ERROR
- ENDKEYs
- STOP
- QUIT

3.1 ERROR

The ERROR condition represents the majority of run-time failures in OpenEdge. The ERROR condition occurs when:

- The AVM fails to execute ABL code, usually at the statement level. These failures are detected at run time by the AVM which then raises the ERROR condition. These errors are known as system errors. A system error is associated with a number and a descriptive message.
- Your application executes the RETURN ERROR statement. An error raised in this way is called an application error.
- An application user presses the key mapped to the ERROR keycode

3.2 ENDKEY

The ENDKEY condition represents run-time occurrences that have significant meaning for an older style of programming that features tightly coupled UI and database record manipulation. The ENDKEY condition is not frequently used in modern applications.

- An application user presses a key that is mapped to the ENDKEY keycode when input is enabled. In character mode applications, input is always enabled.
- The application reaches the end of an input stream.

3.3 STOP

The STOP condition represents a run-time occurrence that requires an application to end the session. Default handling for the STOP condition includes rolling back all current transactions.

The STOP condition occurs when:

- Your application executes a STOP statement.
- An application user presses the key mapped to the STOP keycode when input is enabled.
- The AVM encounters a system error that is deemed unrecoverable.

The AVM also automatically
raises the STOP condition when an unrecoverable system error occurs; for example, when a database connection is lost or a RUN statement cannot find a procedure file.

This **STOP** key is usually mapped as follows:

- **CTRL**–**BREAK** (Windows)
- **CTRL**–**C** (UNIX)

When the STOP condition occurs, by default, the AVM follows these steps:

- If the STOP condition results from an unrecoverable system error, such as a RUN statement that specifies a non-existent procedure file, the AVM displays an error message to the standard output device, which is usually the terminal screen. The NO-ERROR option does not suppress error messages resulting from a STOP condition.
- The AVM undoes the current transaction
- The AVM looks for an ON STOP phrase
- If the application was started from a tool such as the OpenEdge Editor, it returns to that tool. Otherwise, if you used the Startup Procedure (-p) parameter to start the ABL session, and if the startup procedure is still active, the AVM restarts the startup procedure

The STOP statement allows you to raise the STOP condition. The statement executes the default STOP condition handling. Unless you have coded an ON STOP phrase, the STOP statement stops all currently active procedures.

3.4 QUIT

The QUIT condition only occurs when your application executes a QUIT statement.
When the QUIT condition occurs, by default, the AVM follows these steps:

- Commits the current transaction

- If the application was started from the Procedure Editor or Architect, the AVM returns to that tool; otherwise it returns to the operating system.

4 Traditional error handling

4.1 Default error handling

In ABL, when the AVM is not able to properly execute a statement, the AVM most often raises the ERROR condition. If the statement failure is serious enough, the AVM may raise the STOP condition instead. For example, a RUN statement that cannot find a specified procedure file raises the STOP condition. The design of each ABL statement includes decisions about how to respond to run-time conditions.

Default error handling is defined at the block level and is considered a property of the block.
When it fails, the AVM performs the following steps:

- The AVM stops execution of the block at the failed statement.
- The AVM displays a message to the default output destination, which is usually the screen.
- The AVM checks the block that encloses the failed statement to get the default error handling instructions.
- Error handling always begins with the UNDO action. The AVM undoes any changes to undoable variables and temp-table fields. Variables and temp-table fields are undoable by default unless they were defined with the NO-UNDO option. The AVM also undoes the current transaction, if one exists.
- The AVM attempts the default branching option. This could be RETRY, LEAVE, or NEXT, depending on the context.
- Control returns to the caller of the procedure block. If there is no caller, the "application" terminates. The AVM terminates the procedure and returns to the operating system or the OpenEdge tool from which the procedure was run.

4.2 UNDO concept

Transactions are scoped to blocks. Default error handling is also scoped to blocks, and the first step in default error handling is to undo. Thus, undoing a transaction might seem to be synonymous with undoing a block. Actually, the only work being undone is changes to database fields, undoable variables, and temp-table fields.

4.3 Changing block error handling

The ON ERROR phrase lets you change the error handling for a DO, FOR, or REPEAT block. The ON ERROR phrase is the ABL tool for altering the default error handling for basic blocks.

```
DO ON ERROR UNDO, RETURN:
        FIND FIRST Customer WHERE CustNum = 1000.
END.
```

RETRY is the default branching option if you do not use an explicit LEAVE, NEXT, RETRY, or RETURN option. Because RETRY in a block without user input results in an infinite loop, the AVM automatically checks for this possibility and converts a RETRY block into a LEAVE block or a NEXT block if it is an iterating block. This behavior is often referred to as infinite loop protection.

4.4 Suppressing the ERROR condition

Many ABL statements support the NO-ERROR option. The NO-ERROR option directs the AVM to redirect any failure that would have otherwise raised ERROR.

- No message is displayed to the default output
- Execution will continue with the next statement
- Control is not redirected to the block error handling

```
FIND FIRST Customer WHERE CustNum = 1000 NO-ERROR.
IF ERROR-STATUS:ERROR = TRUE THEN
        MESSAGE ERROR-STATUS:GET-MESSAGE(1).
```

Information about errors that occurred on a statement with the NO-ERROR option are redirected to the ERROR-STATUS system handle. The NO-ERROR option redirects errors to the ERROR-STATUS system handle. The handle preserves each system error message raised by a statement with the NO-ERROR option. The handle preserves this information only until the AVM executes another statement with the NO-ERROR option, whether or not an error occurred on the subsequent statement.

If the ABL statement uses the NO-ERROR option and the AVM raises the ERROR condition, then this attribute is set to TRUE. If a method on an ABL system object generates an error message, the AVM does not raise error and this attribute remains FALSE. ABL treats built-in method errors as warnings rather than true errors.

```
DEFINE VARIABLE hSocket AS HANDLE.
CREATE SOCKET hSocket.
hSocket:CONNECT ("-H localhost -S 3333") NO-ERROR.
IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
        RUN FailedSocketConnect.p.
```

4.5 Custom error handling

While the NO-ERROR option can be used to suppress errors you want to ignore, it is more frequently used when you want to:

- Override the default error handling
- Suppress your customer from seeing the default error message
- Provide custom code to handle the particular system error
- Test the ERROR-STATUS system handle to determine if a particular error occurred

4.6 Application error handling

You can raise the ERROR condition yourself with the RETURN ERROR statement and force the default error handling of the block to execute.

- The default error handling occurs after execution resumes in the caller.
- You can also specify a character string with RETURN ERROR that represents your application error data.
- You access the returned string with the RETURN-VALUE function.
- Causes an ERROR condition in the calling block.
- You can use the ERROR option in a procedure, database trigger block, class-based method, constructor, property accessor method. However, you **cannot** use the ERROR option in a user-interface trigger block or destructor to raise ERROR outside of the trigger block.
- Any values that are set for OUTPUT or INPUT-OUTPUT parameters before the RETURN ERROR executes are not returned to the caller.
- RETURN ERROR in a user-defined function does not raise error in the caller. Instead, it sets the target variable of the function to the unknown value. Therefore, you could perform error checking on a function call by checking for the unknown value after a function call. This technique only works if the function uses the RETURN ERROR

statement and the target variable has a value other than the unknown value at the time of the function call.

```
DEFINE VARIABLE iFuncReturn AS INTEGER INITIAL 99 NO-UNDO.

FUNCTION ErrorTest RETURNS INTEGER:
    RETURN ERROR.
END FUNCTION.

ASSIGN iFuncReturn = ErrorTest().

IF iFuncReturn EQ ? THEN
    DISPLAY "Error in user-defined function.".
```

5 Structured Error Handling

5.1 Introduction

Whether a statement raises error is not normally dependent on which error handling model is in effect in a block. However, the presence of a CATCH block in any block allows all errors to be handled more consistently, since it can handle errors that can not be handled by the NO-ERROR option or ON ERROR phrase. In traditional error handling, methods on built-in system handles treat errors as warnings. Therefore, built-in methods do not raise ERROR. If a CATCH block is present in the block containing a failed built-in method, the AVM will now raise ERROR.

System error messages are the same. Because they are wrapped by error objects, you access the messages by properties and methods on the error object, as opposed to attributes and methods on the ERROR-STATUS system handle.

For explicit error handling, the ON ERROR phrase and UNDO statement all support the THROW option. THROW suppresses display of default error messages, exits the block and passes the error raised in the block to the immediate outer block.

When a CATCH block for a particular error type is in effect, the AVM will not display an error message. The error message is instead placed within the error object created by the AVM. If there is no CATCH block for the particular error type, then the AVM will display an error to the standard output.

The NO-ERROR option is an important part of structured error handling. It is the only option that can prevent a statement within a block from raising ERROR where it otherwise would. Even when a CATCH block is present that would handle the error a statement raises, if the NO-ERROR option is used on the statement, the CATCH block ignores it.

If you use the NO-ERROR option to prevent a statement from being handled by a CATCH block, then the ERROR-STATUS system handle continues to be the tool you use to determine if an error occurred on the statement and what the error or errors were.

User-defined functions do not raise ERROR after a RETURN ERROR statement. However, using the THROW directive you can do this with structured error handling.

The FINALLY end block is a useful ABL feature for more than just error handling. The FINALLY block executes at the completion of each iteration of a block whether the execution was successful or raised ERROR.

ABL structured error handling includes the following language additions:

- Adds an extensible hierarchy of classes to allow OpenEdge system errors and your custom application errors to be expressed as objects
- Adds a CATCH
- Adds the UNDO, THROW directive to the ON ERROR phrase
- Adds the THROW option to the UNDO statement
- Adds the ROUTINE-LEVEL ON ERROR UNDO, THROW statement
- Adds another new end block with the FINALLY statement

5.2 ABL Error Classes

The first characteristic of structured error handling is that errors are represented as objects.

5.2.1 Progress.Lang.Error interface

The Progress.Lang.Error interface describes a common set of properties and methods that built-in ABL error classes implement to interact with the ABL structured error handling model. This interface cannot be implemented by a user-defined class. Instead, to create your own ABL error class, subclass the Progress.Lang.AppError class.

Specifying Progress.Lang.Error interface in a CATCH statement, creates an error handler that catches all possible errors.

5.2.2 Progress.Lang.ProError class

Progress.Lang.ProError is the ultimate super class for all ABL built-in and user-defined classes that represent errors in the ABL structured error handling model. You cannot directly inherit from this class. Instead, the immediate subclasses of this class represent the two major types of classes in ABL:

- Progress.Lang.SysError represents any error generated by the AVM
- Progress.Lang.AppError represents any error your application defines

5.2.3 Progress.Lang.SysError class

When an ABL statement raises the error condition, the AVM throws an error. These errors are represented by the Progress.Lang.SysError class. Progress.Lang.SysError inherits generic error handling abilities from Progress.Lang.ProError. You cannot inherit from this class and the class constructors are reserved for system use only.

5.2.4 Progress.Lang.AppError class

Progress.Lang.AppError is the ultimate super class of all application errors. An application error is simply any collection of data you need to provide necessary information about a condition. Representing a user-defined error as an error object allows your application to throw and catch or return the error in the ABL structured error handling model.
When the AVM encounters the RETURN ERROR statement, it implicitly throws a Progress.Lang.AppError error object and places any specified error string in the object's ReturnValue property.
Progress.Lang.AppError class provides the ability to define error objects at run time. In ABL, user-defined error types are called application errors. The Progress.Lang.AppError class lets you define application errors that are handled like ABL system errors. In this paradigm, an error object consists of one or more descriptive CHARACTER messages and (optionally) one or more INTEGER message numbers.

5.3 Handling Errors with CATCH Blocks

Structured error handling allows you to provide custom error-handling code for any type of error. ABL provides the new CATCH statement to handle specific error types. The CATCH statement defines an end block of code that only executes if the ERROR condition is raised in its associated block and the type of error raised is the error type specified in the CATCH statement.

The CATCH block executes once for each iteration of its associated block that raises a compatible error. A block can have multiple CATCH blocks, and all must come at the end of the associated block.

There can only be one CATCH block for each specific error type in a block. However, a CATCH block for a specific error type will also handle error objects for its subtypes. So, it is possible that there can be more than one CATCH block to handle a particular error type in a block. If multiple CATCH blocks are compatible with the error raised, the AVM will execute the first CATCH block it

encounters that is compatible with the error. For this reason, CATCH blocks should be arranged from the most specific error type to the most general error type.

ABL issues a compile-time error if a CATCH block is present in a simple DO block, since simple DO blocks do not have error handling capabilities. DO blocks must have either a TRANSACTION or an ON ERROR directive in order to have a CATCH.

If error is raised in a block and is not handled by a CATCH block, then the error is handled by the ON ERROR directive of the associated block. This could be an explicit ON ERROR phrase, or the implicit (default) ON ERROR phrase for the block type.

One or more CATCH blocks are positioned at the end of the associated block. If a FINALLY block is also used, the CATCH block comes before the FINALLY block.

5.3.1 UNDO scope and relationship to a CATCH block

At the point the CATCH block executes, several steps in the error handling process are already complete, and this state affects what data is available to your CATCH block:

- Any transaction within the associated block will already be undone. In other words, changes made within the associated block to persistent data, undo variables, and undo temp-table fields have already been discarded.
- Records scoped to the associated block will be released before the CATCH block executes. Even if no transaction is present, undoable variables and temp-table fields changed by the associated block will be restored to their last valid values before the associated block began its execution.

5.4 THROW with error objects

As part of the ON ERROR phrase, UNDO, THROW causes the AVM to undo the current transaction, suppress the display of system error messages, create and populate an error object, and throw that error object. Assuming that the errors are not handled within the current block, THROW directs the block enclosing the current block to handle the errors. This ability to direct a containing context to handle errors raised in an inner context is often described as propagating errors up the call stack.

5.5 THROW as a default for routine-level blocks

The undoable blocks (DO, FOR, and REPEAT), support the explicit ON ERROR UNDO, THROW phrase. This phrase is useful for propagating errors up the call stack where they can be handled by CATCH blocks associated with higher level blocks. This technique eliminates the need for CATCH blocks handling common error types at every level in a series of nested blocks.

The main blocks of ABL routines (routine-level blocks) do not support explicit ON ERROR phrases. The routine-level blocks have an implicit ON ERROR UNDO, RETRY phrase when user input is detected in the block and an implicit ON ERROR UNDO, LEAVE when no user input is detected. You can use the ROUTINE-LEVEL ON ERROR UNDO, THROW statement in a procedure (.p) or class (.cls) file to change the implicit ON ERROR phrase associated with routine-level blocks.

5.6 THROW with user-defined functions

The user-defined function, which is defined by the FUNCTION statement, returns a value of a specific data type as its primary function. The RETURN statement is the statement you use to specify in the function body what value to return to the caller. This fact makes it impossible to use the RETURN ERROR statement in the same way as it is used in other blocks. RETURN ERROR in a user-defined function does not raise error in the caller. Instead, it sets the target variable of the function to the unknown value. Therefore, you could perform error checking on a function call by checking for the unknown value after a function call. This technique only works if the function uses the RETURN ERROR statement and the target variable has a value other than the unknown value at the time of the function call. With structured error handling, you can raise error in the caller by throwing an error from a CATCH block on the main function block. A CATCH block in the caller can then handle system
Error.

5.7 Using FINALLY End Blocks

In object-oriented programming, the importance of clean-up code that destroys unneeded objects and frees up other resources is vital. The FINALLY statement supports such maintenance tasks. The FINALLY statement creates an end block that executes once at the end of each iteration of its associated block, whether or not the associated block executed successfully or raised the ERROR condition.

The FINALLY block executes after:

- Successful execution of the associated block

- Each successful iteration of an iterating associated block
- ERROR is raised in the associated block and a CATCH block handles the error
- ERROR is raised in the associated block and no CATCH block handles the error

The FINALLY block will not execute if:

- A STOP condition is raised and not handled
- A QUIT statement is in effect and it is not handleds

There can only be one FINALLY block in any associated block. The FINALLY statement must come after all other executable statements in the associated block. If the associated block contains CATCH statements, the FINALLY block must come after all CATCH blocks. Note that the FINALLY statement **can** be used in a block with no CATCH blocks.

The purpose of a FINALLY block is to hold clean-up code that must execute regardless of what else executed in the associated block. It can include code to delete dynamic objects, write to logs, close outputs, and other routine tasks. Because it executes even if the ERROR condition is raised, the FINALLY block is also a useful part of a structured error handling scheme.
If the AVM detects a STOP or QUIT condition in the associated block, the FINALLY block will not run and the AVM processes the condition. If the associated block has an ON STOP or ON QUIT phrase, then the STOP or QUIT condition is handled and released by the time the AVM is ready to execute the FINALLY block, and the FINALLY block is executed.