Progress®



OpenEdge® Development:
ABL Error Handling

OpenEdge®

Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

February 2019

Last updated with new content: Release 12.0

Updated: 2019/03/01

Table of Contents

Preface....9
Purpose....9
Audience....9
Organization....10
Using ABL documentation....10
References to ABL compiler and run-time features....11
References to ABL data types....11
Typographical conventions....11
Examples of syntax descriptions....12
Long syntax descriptions split across lines....14
Complex syntax descriptions with both required and optional elements....14
OpenEdge messages....15
Obtaining more information about OpenEdge messages....15

Chapter 1: Introduction to error and condition handling....17
ABL conditions....17
ABL condition handling....19
Terminology....19

Chapter 2: Default Condition Handling....21
Understanding OpenEdge messages....22
Progress messages (promsgs) file....22
Understanding the UNDO concept....22
Branching options....23

Chapter 3: Using NO-ERROR....25
NO-ERROR behavior....25
Precedence of NO-ERROR....26
Using NO-ERROR to trap a thrown object....26
UNDO and scope using NO-ERROR....27
Handling the error....27
Handling warnings....28
Incorrect use of NO-ERROR....29

Chapter 4: Block Flow of Control and Condition Directives....31
ON phrase syntax....33

Using labels....35
Precedence of the ON phrase....36
Examples using ON ERROR....37
Using UNDO, THROW....38
BLOCK-LEVEL ON ERROR UNDO, THROW statement....38
ROUTINE-LEVEL ON ERROR UNDO, THROW statement....40
-undothrow startup parameter....41
Determining error-handling characteristics of r-code....41
ON phrases and STOP conditions....42
Throwing error and stop objects from an AppServer to an ABL client....42

Chapter 5: ERROR and STOP Classes....45

Progress.Lang.Error interface....47
Progress.Lang.ProError class....48
Progress.Lang.SysError class....48
Progress.Lang.StopError class....48
Progress.Lang.SoapFaultError class....48
Progress.Lang.Stop class....49
Progress.Lang.StopAfter class....49
Progress.Lang.UserInterrupt class....49
Progress.Lang.LockConflict class....50
Progress.Lang.AppError class....50
AppError Constructors....51
.NET Exceptions....53
Enabling stack tracing with error objects....53

Chapter 6: Raising Conditions....55

Raising errors with UNDO, THROW....55
RETURN ERROR....57
Raising ERROR to the caller of a user-defined function....59
Throwing a condition out of a destructor....60
Raising the QUIT condition....61
Raising the STOP condition....61
Raising a timed STOP condition....62
Throwing error and stop objects from an AppServer to an ABL client....62

Chapter 7: CATCH Blocks....65

Introduction to CATCH blocks....65
CATCH block syntax and usage....67
Blocks that support CATCH blocks....70
Precedence of CATCH blocks....70
UNDO scope and relationship to a CATCH block....71
CATCH blocks within CATCH blocks....73

Chapter 8: FINALLY Blocks.....75
Introduction to FINALLY blocks.....75
FINALLY block syntax and usage.....76
UNDO scope and relationship to a FINALLY block.....76
Examples using FINALLY blocks.....77
FINALLY blocks and STOP-AFTER.....79
Conflicts between the associated and FINALLY blocks.....79

Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Using ABL documentation
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages

Purpose

This book provides a programming guide for handling ABL *conditions* (also known as *errors* or *exceptions*). Conditions are run-time occurrences that interrupt the usual flow of a software application. In ABL, conditions include ERROR, STOP, QUIT, and ENDKEY, all of which are ABL keywords.

Audience

This book is intended for all ABL programmers.

To understand the material in this manual, you should be familiar with the following ABL topics:

- Procedure files (.p), internal procedures, and user-defined functions

- Blocks and block properties
- Transactions
- Built-in system objects, attributes, and methods

OpenEdge Getting Started: ABL Essentials is your best resource for learning about these topics. In particular, the chapters on transactions demonstrate the relationship between transactions, blocks, block properties, and default error handling.

Organization

Introduction to error and condition handling on page 17

Provides an overview of the terminology used for ABL errors/conditions/exceptions, and the constructs for handling them.

Default Condition Handling on page 21

Describes the default behavior when an error, or condition, occurs.

Using NO-ERROR on page 25

Describes how and why to use the `NO-ERROR` option on an ABL statement.

Block Flow of Control and Condition Directives on page 31

Documents the use of the `ON` phrase for altering the default error handling.

ERROR and STOP Classes on page 45

Describes the hierarchy of built-in ABL classes that represent `ERROR` and `STOP` conditions.

Raising Conditions on page 55

Discusses ABL constructs for raising conditions programmatically.

CATCH Blocks on page 65

Provides in-depth information on using `CATCH` blocks to handle errors.

FINALLY Blocks on page 75

Describes how to use the `FINALLY` block for end-of-block processing.

Using ABL documentation

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is *ABL (Advanced Business Language)*. With few exceptions, all keywords of the language appear in all `UPPERCASE`, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates go to https://docs.progress.com.

References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the *ABL Virtual Machine (AVM)*. When the documentation refers to ABL source code compilation, it specifies *ABL* or *the compiler* as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies *the AVM* as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time: "ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error." The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: "The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all `UPPERCASE`, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever *integer* appears, this is a reference to the `INTEGER` or `INT64` data type.
- Wherever *character* appears, this is a reference to the `CHARACTER`, `LONGCHAR`, or `CLOB` data type.
- Wherever *decimal* appears, this is a reference to the `DECIMAL` data type.
- Wherever *numeric* appears, this is a reference to the `INTEGER`, `INT64`, or `DECIMAL` data type.

References to built-in class data types appear in mixed case with initial caps, for example, `Progress.Lang.Object`. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This documentation uses the following typographical and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.

Convention	Description
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax, code examples, system output, and file names.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax.
`Fixed-width bold`	Fixed-width bold italic indicates variables in syntax with special emphasis.
`UPPERCASE fixed width`	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
Period (.) or colon (:)	All statements except `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` end with a period. `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

Examples of syntax descriptions

In this example, `ACCUM` is a keyword, and *`aggregate`* and *`expression`* are variables:

Syntax

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
  DISPLAY Customer.Name.
END.
```

In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

Syntax

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

Syntax

```
INITIAL [ constant [ , constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

Syntax

```
{ &argument-name }
```

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

Syntax

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

Syntax

```
MAXIMUM ( expression , expression [ , expression ] ... )
```

In this example, you must specify MESSAGE and at least one *expression* or SKIP [(*n*)], and any number of additional *expression* or SKIP [(*n*)] is allowed:

Syntax

```
MESSAGE { expression | SKIP [ ( n ) ] } ...
```

In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

Syntax

```
{ include-file
    [ argument | &argument-name = "argument-value" ] ... }
```

Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, WITH is followed by six optional items:

Syntax

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more *field* entries or one *record*. Options available with *field* or *record* are grouped with braces and brackets:

Syntax

```
ASSIGN   { [ FRAME frame ] { field [ = expression ] }
           [ WHEN expression ] } . . .
       | { record [ EXCEPT field . . . ] }
```

OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.
- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.
- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.
- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is `200`:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help** > **Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help** > **Messages** and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the OpenEdge `pro` command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

```
OpenEdge-install-dir/bin/pro
```

2. Press **F3** to access the menu bar, then choose **Help** > **Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File** > **Exit**.

1

Introduction to error and condition handling

Run-time occurrences that interrupt the usual flow of a software application are called *conditions*, *errors*, or *exceptions*. *Condition handling*, *error handling*, and *exception handling* are industry terms, sometimes used interchangeably, for programming designed to respond to those run-time interruptions. Since most condition handling in ABL involves the `ERROR` condition, *error handling* has become a common synonym for *condition handling* and is used frequently in the OpenEdge documentation set.

For details, see the following topics:

- ABL conditions
- ABL condition handling
- Terminology

ABL conditions

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, which is called *default error handling*.

Note: An ERROR is the most common type of condition, which is why *condition handling* is usually referred to as *error handling* in ABL documentation.

Default error handling protects your database while also providing branching options to restore application flow. You design your ABL application to accept default error handling, or to replace it with custom error handling.

ABL recognizes these four conditions, all of which are keywords:

- ERROR

  An ERROR occurs when:

  - The AVM fails to execute an ABL statement. For example, if a FIND statement fails to find a matching record, then the statement fails. 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 or throws an instance of Progress.Lang.AppError. An error raised in this way is called an *application error*.
  - An application uses the APPLY "ERROR" statement. This is considered outdated functionality.

  In all cases, when an error condition is raised in a block with error handling, it is not raised again beyond that block unless some ABL code explicitly instructs the AVM to do so. In other words, the default is that errors are local and are not propagated beyond the current block context.

- STOP

  The STOP condition represents a more serious condition and occurs when:

  - The AVM encounters a system error that is deemed more serious or unrecoverable. For example, if the AVM detects a lost database connection, the AVM raises the STOP condition.
  - Your application executes a STOP statement.
  - An application user presses the key mapped to the STOP key code when input is enabled. By default, this is **CTRL+C** on Unix/Linux or **CTRL+Break** in Windows.
  - The specified time has expired when using a STOP-AFTER phrase on a block.
  - A record lock conflict occurs and the user either cancels out of the ensuing record lock conflict dialog, or there is no user response (including in batch mode) within the time specified by the -lkwtmo startup parameter.

  In all cases, when a STOP condition is raised in a block, it is then raised again in all blocks up the call stack unless it is handled by explicit ABL code. In other words, the default is that STOP conditions propagate up the call stack indefinitely until the application ends. In addition, there are a few cases where the AVM ignores explicit attempts to handle a STOP condition, until either the current transaction is over or until the AVM has returned up past the code layer that accesses the database, depending on the condition.

- QUIT

  The QUIT condition only occurs when your application executes a QUIT statement. It causes any open transaction to be committed and the AVM session to terminate. As the AVM unwinds the stack, the QUIT condition is in effect.

- ENDKEY

  The ENDKEY condition occurs when:

  - An application user presses a key that is mapped to the ENDKEY key code (typically **ESC**) when input is enabled. This has different default behavior depending on the context. For example, it may back out editing entered by a user in an ABL widget, or it may close the current window if no editing is taking place. This style of user interface was designed for character mode applications and is no longer relevant to most modern applications.
  - The application reaches the end of an input stream.

ABL condition handling

Default condition handling exists for any condition that occurs in an application. This is explained in more detail in Default Condition Handling on page 21. However for most applications, the default handling does not provide the most desirable way to control program flow or show error messages in a user-friendly manner. Therefore, most applications customize how conditions are handled.

Below is a summary of the various ABL constructs that can be used to customize condition handling. Each is discussed in more detail in later sections.

- The `NO-ERROR` keyword — This is useful when an error is expected on a single statement and there is a specific programmatic way of handling it.
- The `ON` phrase on a block — This includes `ON ERROR`, `ON STOP`, `ON QUIT`, and `ON ENDKEY`. These are used to determine program flow after an unexpected condition, or to throw the condition to a higher level to be handled there.
- `CATCH` blocks — This is the most desirable way to deal with unexpected `ERROR` and `STOP` conditions. It gives the program access to error information so that it can be reported in a customized way.
- Error objects — The ABL provides a set of built-in objects representing `ERROR` or `STOP` conditions, which can be caught. In addition, applications can create their own application error objects that can be thrown and may provide extra contextual information.
- The `UNDO, THROW` statement — This is a way to rethrow a caught error or stop object, or throw an application error object.
- `FINALLY` blocks — This is not specifically about handling conditions. However, it allows you to write code that always runs at the end of a block, whether a condition occurs in that block or not.

Terminology

The original ABL condition handling model consisted of the following subset of the existing error handling constructs:

- `NO-ERROR`
- The `ON` phrase (flow of control directives), but without the `THROW` option
- `RETURN ERROR`

These constructs are referred to as *Traditional Error Handling*.

In more recent OpenEdge versions, more modern error handling constructs were added. These consist of:

- `CATCH` blocks
- The `ON THROW` directive
- The `UNDO, THROW` statement
- Error and Stop objects
- `FINALLY` blocks

These constructs are referred to collectively as *Structured Error Handling*.

These two models are not independent of each other. All constructs work seamlessly together, and in some cases, depend on each other for context.

2

Default Condition Handling

The default for handling conditions in ABL is simple. It is scoped to blocks. Several things happen when an error occurs, when not explicitly handled by some ABL construct such as `NO-ERROR` or `CATCH`:

1. If there is any error message associated with the condition, it is displayed to the current output device. For an AppServer, the message is written to the AppServer log.
2. The current block is undone. If the condition occurred in an iterating block, it is only the current block iteration that is undone.
3. Any statements following the line where the condition occurred are not executed. Program flow continues based on the default branching option for the block type.

   Note: If there is a `FINALLY` block associated with the current block, it still runs even though an `ERROR` or `STOP` condition occurred. For more information see FINALLY Blocks on page 75.

4. For a `STOP` condition, `STOP` is raised again in the outer block, if there is one, otherwise it is raised in the caller. From there the same actions (1-4) are taken with respect to the new current block.

For details, see the following topics:

- Understanding OpenEdge messages
- Understanding the UNDO concept
- Branching options

Understanding OpenEdge messages

Many error messages are generated by the AVM itself. These run-time system error messages give information about what failed when an ERROR or STOP condition is raised. Error messages can also be specified by the application itself. For the remainder of this document, the term *error message* refers to either an OpenEdge system error message (execution message), or a user-defined application error message. However, this section specifically discusses what is in a system error message.

An OpenEdge error is associated with an error string and an error number, as shown in the following example:

```
Error string (Error number)

**FIND FIRST/LAST failed for table Customer (565)
```

In this example, "**FIND FIRST/LAST failed for table Customer" is the error string, and "565" is the error number. In OpenEdge documentation, the term *error message* refers to the error string and error number together.

To access additional information for obtaining specific error messages through the Help system see Obtaining more information about OpenEdge messages on page 15.

Progress messages (promsgs) file

The OpenEdge platform stores its messages in a promsgs (Progress messages) file. The promsgs file is available in multiple languages. Setup options for this file are described in *Configure OpenEdge*. You can find information on translation and localization of promsgs files in *Internationalize ABL Applications*.

Understanding the UNDO concept

The ABL UNDO action ensures pending changes to persistent data (database fields) are not committed to a database after an ERROR or STOP condition occurs. Because ABL is transaction-oriented, a set of pending changes is equivalent to an open (current) transaction or subtransaction. *Undoing* is essentially throwing away the current transaction or subtransaction.

ABL also extends UNDO protection to non persistent data like variables and temp-table fields. By default, ABL makes variables and temp-table fields undoable. If changes to undoable variable data occur in a block, the AVM undoes changes to these variables and fields, **but only if this block is a transaction block**.

A block is a transaction block if it contains one of the following statements with a reference to a **database** field:

- CREATE
- DELETE
- ASSIGN (and the = operator)
- INSERT
- SET
- UPDATE

- Statements that fetch database records with EXCLUSIVE-LOCK

If the block statement uses the TRANSACTION option, it is also a transaction block. One use case for this option is to force ABL to create a transaction for undoable variables and temp-table fields when the block does not also update database fields. You can use the COMPILE statement listing options to see which blocks in your code are transaction blocks.

Since providing UNDO behavior for variable and temp-table data incurs additional overhead, it's best to define variables and temp-tables fields with the NO-UNDO option when possible. With the NO-UNDO option, the AVM does not allocate the resources needed to track changes, and any UNDO action ignores the NO-UNDO data items.

Actions other than changes to database fields, undoable variables, and temp-table fields are not affected. For example, if you opened a file or a query within the block, undoing the block does not return the file or query to its closed state.

Note: This introduction to UNDO touches on related transaction concepts. Understanding transactions is an important prerequisite to understanding default error handling. Transaction information in this section describes some default transaction behavior and presumes simple use cases. You should have a good understanding of how to define transactions and subtransactions to accurately model your business logic before continuing. For more information, see the section on managing transactions in *Develop ABL Applications*.

Branching options

After a block performs its UNDO operation, the AVM must determine what action to take next. ABL has the following set of branching (flow of control) options:

- RETRY — If a block is an iterating block, the RETRY action repeats the iteration of the block. RETRY is useful when you want to give your users another chance to input correct data.
- LEAVE — Indicates that the AVM should exit the block and resume execution with the next statement.
- NEXT — Indicates that the AVM should exit the current iteration of a block and continue with the next iteration. If there is not another iteration, then NEXT is the same as LEAVE.
- RETURN — Indicates that the AVM should exit the block and immediately exit the current routine. Execution resumes in the caller. If no caller exists, then the application terminates. The RETURN statement has many options and is discussed from an error handling perspective in RETURN ERROR on page 57.
- THROW — Indicates that the AVM should capture any error message in an error or stop object, exit the block, and raise the same condition again in the next enclosing block, if there is one, otherwise in the caller. The thrown object is then available in the outer block to be handled there.

Each block type has default branching behavior. The following table lists the default action by block type and by context.

Table 1: Default branching for ERROR conditions

Block Type	Action if user input detected	Action otherwise
DO TRANSACTION	RETRY	LEAVE
FOR EACH	RETRY	NEXT

Block Type	Action if user input detected	Action otherwise
REPEAT	RETRY	LEAVE
CATCH	THROW	THROW
FINALLY	THROW	THROW
Routine-level blocks (for example, procedures, methods)	RETRY	LEAVE
Trigger procedure file	RETURN ERROR	RETURN ERROR

Table 2: Default branching for STOP conditions

Block Type	Action
DO TRANSACTION	LEAVE
FOR EACH	LEAVE
REPEAT	LEAVE
CATCH	LEAVE
FINALLY	LEAVE
Routine-level blocks (for example, procedures, methods)	LEAVE
Trigger procedure file	LEAVE

For STOP conditions, the block action is to leave, but the condition is raised again in the outer, or calling block, by default. Even though the condition is raised again, this is not the same thing as a THROW. With THROW, the error message is trapped in an object and is only displayed if you catch it and display the message itself, or if the error raised at the outer/upper level is not handled or thrown again. It is not displayed at the statement where the error occurred. For STOP, the default is to display any error message immediately, leave the block, and raise the condition again in the outer/upper block. The message is only displayed once at the place where the STOP condition occurred.

3

Using NO-ERROR

One way to handle an error condition is to use the NO-ERROR option on specific ABL statements. This only applies to ERROR conditions, not to STOP or QUIT conditions. It should be used when an error might be expected from a specific statement and you can take some programmatic action when the expected error occurs.

A common example is using NO-ERROR on the FIND statement. You might be looking for a record based on certain criteria and there may, or may not, be any records that satisfy that criteria. If there are not, you can modify the criteria and try the FIND again, or you can inform the user to pick a different option to search on. In either case, there is an action to be taken based on this specific ERROR condition.

For details, see the following topics:

- NO-ERROR behavior
- Precedence of NO-ERROR
- Using NO-ERROR to trap a thrown object
- UNDO and scope using NO-ERROR
- Handling the error
- Handling warnings
- Incorrect use of NO-ERROR

NO-ERROR behavior

When an error occurs on a statement that uses the NO-ERROR option, the AVM takes the following actions:

- Any error messages generated by the statement are not displayed to the default output. Instead they are redirected to a system handle called ERROR-STATUS.

  The handle preserves all system error messages raised by the statement, setting the NUM-MESSAGES attribute accordingly. 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. This is illustrated further in the next section, Precedence of NO-ERROR on page 26.

- If any undo-able action has already occurred as part of the statement, that action is undone. This is discussed in more detail in the section, UNDO and scope using NO-ERROR on page 27.

- Execution continues with the next statement.

Note: The NO-ERROR option has no effect on STOP condition handling.

Refer to the *ABL Reference* to see specific statements that support the NO-ERROR option.

Precedence of NO-ERROR

NO-ERROR takes precedence over any flow of control directive on the block, for example, LEAVE or THROW. See Default Condition Handling on page 21 and Block Flow of Control and Condition Directives on page 31 for more information.

It also takes precedence over any CATCH blocks so the CATCH block does not run. See CATCH Blocks on page 65 for more information.

In general, the AVM performs error handling using this precedence, from highest to lowest. The AVM only abides by one of these when a condition is raised:

- Statement NO-ERROR option
- CATCH block
- Block's ON phrase (explicit or implicit)

Using NO-ERROR to trap a thrown object

System errors may be generated directly by the AVM or they may be caught as an object and rethrown. You can also throw an instance of a custom application error object. Both techniques are discussed in detail in Raising Conditions on page 55.

In either case, you can choose to handle the thrown object by using NO-ERROR. Any error messages in the object are transferred to the ERROR-STATUS system handle, the ERROR-STATUS:ERROR is set to TRUE, and the error object instance is garbage collected. Any custom information that might have been in the error object is lost. You can then handle the information in the same way that you would for a condition generated any other way.

UNDO and scope using NO-ERROR

NO-ERROR is a statement-based construct. If the statement where NO-ERROR is used can update database fields or undoable program variables (for example, the ASSIGN statement), the AVM creates a subtransaction around the statement. This means that any modifications that occur before the error happens are undone. In this scenario, if the statement includes an expression that contains other executable elements, like method calls, the undoable operations performed by these elements are also undone, since they are part of the subtransaction.

On the other hand, if the statement itself cannot make any updates to database fields or undoable program variables, the AVM does not start a subtransaction. For example, specifying the NO-ERROR option on a RUN statement does not have any effect on whether statements inside the procedure are undone; that is controlled by constructs inside the procedure itself.

Similarly, the scope of the NO-ERROR option is only the current statement. It only traps information on an error that is raised to the level of that statement. It has no effect, in terms of error message suppression, on statements in sub-blocks that may be invoked by the statement. For example, the FIND statement in the following code displays an error to the current output device. It is not trapped by the NO-ERROR option on the RUN statement.

```
RUN subProcedure NO-ERROR.

PROCEDURE subProcedure:

    /* Nonsense code raises ERROR.*/
    FIND SalesRep WHERE SalesRep.RepName = Customer.Name.
END.
```

Handling the error

The ERROR-STATUS system handle allows you to test whether *any* error occurs and whether a *particular* error occurs. You can have a branch that executes for a particular error, and you can have another branch that executes for any other error.

The attributes and methods of the handle allow you to access the error message strings and error numbers. If specific errors are important to you, the error numbers are useful. But more often than not, you are simply interested in whether an error occurs.

The following table describes the significant attributes and methods of the ERROR-STATUS system handle.

Attribute or method	Description
ERROR attribute	If the ABL statement uses the NO-ERROR option and the AVM raises the ERROR condition, this attribute is set to TRUE. Some handle methods may generate an error message but not raise ERROR. In this case the condition is treated as a warning and the attribute remains FALSE. However ERROR-STATUS:NUM-MESSAGES is still set to a nonzero value. See Handling warnings on page 28 for more detail.

ERROR-OBJECT-DETAIL attribute	If a Web service method returns a SOAP fault, the AVM stores the SOAP fault information in an ABL SOAP-fault object and raises ERROR. The AVM stores a handle reference to the SOAP-fault object in this attribute.
NUM-MESSAGES attribute	Provides an integer count of all the error messages generated by the statement with the NO-ERROR option.
GET-MESSAGE(*index*) method	Allows you to retrieve the specified error string. The index runs from 1 to the value of NUM-MESSAGES.
GET-NUMBER(*index*) method	Allows you to retrieve the specified error number. The index runs from 1 to the value of NUM-MESSAGES.

The following example illustrates using the FIND statement with NO-ERROR:

```
METHOD PUBLIC DECIMAL getCustomerBalance(custName AS CHAR):

    FIND FIRST Customer WHERE Customer.NAME = custName NO-ERROR.
    IF ERROR-STATUS:ERROR THEN
        RETURN ERROR. // No Customer found with that name
    ELSE
        RETURN Customer.Balance.
END.
```

For the FIND statement in particular, there is another way to do this. When the FIND statement fails, the buffer is left with no record in it. Therefore, you can use the built-in AVAILABLE function to determine if the FIND failed. In this case, we don't use the information in the ERROR-STATUS system handle, but still use NO-ERROR to prevent the error message from displaying.

```
METHOD PUBLIC DECIMAL getCustomerBalance(custName AS CHAR):

    FIND FIRST Customer WHERE Customer.NAME = custName NO-ERROR.
    IF AVAILABLE Customer THEN
        RETURN Customer.Balance.
    ELSE
        RETURN ERROR. // No Customer found with that name
END.
```

Handling warnings

The error handling behavior of some handle methods is different depending on whether or not structured error handling is in effect in the block where the method is called. It is in effect if you have a CATCH block and/or you are using the UNDO, THROW directive on the block.

Note: There are several ways to set the UNDO, THROW directive for a block. See Block Flow of Control and Condition Directives on page 31 for more detail.

Without structured error handling, these handle methods do not raise an error when the method fails, even though it generates an error message. The AVM treats the error as if it is a warning. By default, the error message displays to the current output device but execution continues at the next line, as if no error occurred.

In this case, you cannot use the ERROR-STATUS:ERROR attribute to detect that something went wrong. However, the error messages are still saved in the ERROR-STATUS handle. Therefore, you should check NUM-MESSAGES instead, as in this example:

```
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.
```

If there is structured error handling on the block, the method raises an error, not a warning. The above code still works as-is since NUM-MESSAGES is greater than 0 whether it is a warning or an error. But with structured error handling you can alternatively code it by checking for ERROR-STATUS:ERROR, as in this code:

```
DEFINE VARIABLE hSocket AS HANDLE.

DO ON ERROR UNDO, THROW: // for unexpected errors
    CREATE SOCKET hSocket.
    hSocket:CONNECT ("-H localhost -S 3333") NO-ERROR.

    IF ERROR-STATUS:ERROR THEN
        RUN FailedSocketConnect.p.
END.
```

Incorrect use of NO-ERROR

The following example shows incorrect usage of the NO-ERROR phrase. Since the information in the ERROR-STATUS handle is reset each time it is used, whether an error occurs or not, using it on successive statements, and only checking it at the end of the sequence, may cause the application to lose information and behave incorrectly. To trap any error that occurs in a set of statements, use a CATCH block. See CATCH Blocks on page 65 for more information.

```
DEFINE VAR hSocket AS HANDLE.
DEFINE VAR mem AS MEMPTR.

CREATE SOCKET hSocket.
hSocket:CONNECT ("-H localhost -S 3333") NO-ERROR.
FOR EACH Customer WHERE Customer.Name BEGINS "A":
    PUT-STRING(mem, 1) = Customer.NAME NO-ERROR.
    hSocket:WRITE(mem, 1, LENGTH(Customer.Name)) NO-ERROR.
END.
hSocket:DISCONNECT() NO-ERROR.

/* This only tells you if the DISCONNECT call failed. You won't
    even know whether the socket ever connected successfully. */
IF ERROR-STATUS:ERROR THEN DO:
    <Handle the error>
END.
```

4

Block Flow of Control and Condition Directives

The ON phrase is one of the ABL constructs used for altering the default error handling for basic blocks. For information on default error (condition) handling see Default Condition Handling on page 21. There is a variation of the ON phrase to control each of the ABL conditions:

- ON ERROR ...
- ON STOP ...
- ON QUIT ...
- ON ENDKEY ...

The ON phrase can be used on the following blocks:

Block type	Examples
DO (This includes all DO block variations: simple DO blocks, DO TRANSACTION, DO WHILE, and DO FOR)	DO ON STOP UNDO, LEAVE: DO TRANSACTION ON ERROR UNDO, THROW:
FOR EACH	FOR EACH Order ON ERROR UNDO, NEXT:
REPEAT	REPEAT ON QUIT UNDO, LEAVE:

There can be multiple ON phrases on the same block, separated by white space. For example:

```
DO TRANSACTION ON ERROR UNDO, RETRY
               ON STOP UNDO, LEAVE:
END.
```

You cannot modify the error action on other blocks using the ON phrase. This includes:

- Procedures (top-level or internal)
- User-defined methods
- User-defined property getter/setter blocks
- Constructors/destructors
- User-defined functions
- UI trigger blocks
- Database triggers (ON block with CREATE, DELETE, WRITE, or ASSIGN event)

However, there is a way to change the default error handling for most of these blocks, at a block or routine level. This is described in the section on Using UNDO, THROW on page 38.

For details, see the following topics:

- ON phrase syntax
- Using labels
- Precedence of the ON phrase
- Examples using ON ERROR
- Using UNDO, THROW
- ON phrases and STOP conditions
- Throwing error and stop objects from an AppServer to an ABL client

ON phrase syntax

This is the full syntax for the ON phrase. Note there are slightly different options for the different phrases. Specifically, only ON ERROR has the THROW option. STOP conditions are thrown by default, so there is no need to specify TRHOW in the ON STOP syntax. ON ENDKEY and ON QUIT do not have the THROW option since they are older constructs and do not participate in the newer structured error handling model. In addition, ON QUIT does not require the UNDO option, unlike the others.

```
ON ERROR UNDO
  [label1]
  [     , LEAVE [ label2 ]
      | , NEXT [ label2 ]
      | , RETRY [ label1 ]
      | , RETURN [ return-value |
                   ERROR [ return-value | error-object-expression ] |
                   NO-APPLY ]
      | , THROW
  ]
```

```
ON  QUIT [ UNDO [ label1 ] ]
  [     , LEAVE [ label2 ]
      | , NEXT [ label2 ]
      | , RETRY [ label1 ]
      | , RETURN [ return-value |
                   ERROR [ return-value | error-object-expression ] |
                   NO-APPLY ]
  ]
```

```
ON [ STOP | ENDKEY ] UNDO
  [label1]
  [     , LEAVE [ label2 ]
      | , NEXT [ label2 ]
      | , RETRY [ label1 ]
      | , RETURN [ return-value |
                   ERROR [ return-value | error-object-expression ] |
                   NO-APPLY ]
  ]
```

label1

The name of the block whose processing you want to undo. If you do not name a block with *label1*, the AVM undoes the processing of the block started by the statement that contains the ON ERROR/STOP/ENDKEY phrase.

LEAVE [*label2*]

Indicates that after undoing the processing of a block, the AVM leaves the block labeled *label2*. If you do not name a block, the AVM leaves the block labeled with *label1*. There are restrictions. For example, you cannot undo an outer block, but leave only the inner block.

NEXT [*label2*]

Indicates that after undoing the processing of a block, the AVM executes the next iteration of the block you name with the *label2* option. If you do not name a block with the NEXT option, the AVM executes the next iteration of the block that contains the ON phrase.

RETRY [*label1*]

Indicates that after undoing the processing of a block, the AVM repeats the same iteration of the block.

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.

RETURN ...

Returns to the calling routine, if there is one. The following table describes various RETURN options:

Option	Description
return-value	In procedures and VOID methods, this must be a CHARACTER string. The caller can use the RETURN-VALUE function to read the returned value. For user-defined functions, non-VOID methods and property getters, the value must match the specified return type.
ERROR	Undoes the current subtransaction, and raises ERROR in the caller. You cannot specify ERROR within a user-interface trigger block or a destructor. For user-defined functions see note below.

ERROR *return-value*	Undoes the current subtransaction, and raises ERROR in the caller. The CHARACTER string you provide is available to the caller in the RETURN-VALUE function. The AVM also creates an AppError object and stores the *return-value* in the ReturnValue property. For user-defined functions see note below.
ERROR *error-object-expression*	Undoes the current subtransaction, and raises ERROR in the caller. The specified error object instance is thrown to the caller. For user-defined functions see note below.
NO-APPLY	In a user-interface trigger, prevents the AVM from performing the default behavior for the trigger event. Otherwise, the option is ignored.

Note: Using RETURN ERROR in a user-defined function sets the target variable of the function to the Unknown value (?) instead of raising ERROR in the caller. See Raising ERROR to the caller of a user-defined function on page 59 for more detail.

THROW

Use this directive to explicitly propagate an error to the enclosing block, if there is one, otherwise to the caller. You can learn more about throwing error objects in Raising Conditions on page 55.

Using labels

Labels can be used to undo the transaction associated with the outer block, rather than just the subtransaction of the inner block.

The following example sets up a common set of nested FOR EACH blocks that list the order numbers for the first few customer records in the Sports2000 database. Within the inner block, a nonsensical FIND statement raises error after the first iteration. This trivial framework allows you to test the interactions of ON ERROR phrases.

```
PROCEDURE NestedBlocks:

Outer-Block:
    FOR EACH Customer WHERE CustNum < 5:
        ASSIGN Customer.Name = Customer.Name + "_changed".

Inner-Block:
        FOR EACH Order OF Customer
            ON ERROR UNDO Outer-Block, RETURN:

            DISPLAY OrderNum.

            /* Nonsense code raises ERROR. */
            FIND SalesRep WHERE RepName = Customer.Name.

        END. /* Inner-Block */
    END. /* Outer-Block */

    DISPLAY "For Blocks Complete".
END PROCEDURE.

RUN NestedBlocks.

DISPLAY "Procedure NestedBlocks Complete."
```

The flow of this example is as follows:

1. The ASSIGN statement in Outer-Block starts a transaction.
2. The FIND statement in Inner-Block raises the ERROR condition.
3. The error message is displayed.
4. The explicit ON ERROR phrase of Inner-Block activates, causing the entire Outer-Block transaction to be undone, and a RETURN to the main block.
5. The string "Procedure NestedBlocks Complete." is displayed.

Precedence of the ON phrase

ON phrases are at the bottom of the order of precedence for handling errors. If another error handling construct is used, specifically NO-ERROR or CATCH blocks, the ON phrase is ignored. If there are CATCH blocks, but none of them are compatible with the type of condition that occurs, then the ON phrase takes effect (assuming NO-ERROR is not used on the statement).

In general, the AVM performs error handling using this precedence, from highest to lowest. The AVM only abides by one of these when a condition is raised:

- Statement NO-ERROR option
- CATCH block
- Block's ON phrase (explicit or implicit)

Examples using ON ERROR

The following simple code sample illustrates some of the ON phrase constructs.

```
PROCEDURE ScanCustomers:
    DEFINE VAR num AS INTEGER.

    FOR EACH Customer:
        FOR EACH Order OF CUSTOMER ON ERROR UNDO, RETURN:

            /* Nonsense code raises ERROR. */
            Num = INTEGER(Order.BillToID).  //Fails since BillToId is not all numeric

            …
        END.
    END.

    DISPLAY "For blocks complete".
END PROCEDURE.

RUN ScanCustomers.
DISPLAY "Procedure ScanCustomers complete".
```

The following table lists all the ON ERROR phrases in effect in this procedure from the outermost to the innermost.

Block	ON ERROR phrase
Procedure block (.p file)	Implicit ON ERROR UNDO, LEAVE
Internal procedure ScanCustomers	Implicit ON ERROR UNDO, LEAVE
FOR EACH Customer block	Implicit ON ERROR UNDO, NEXT
FOR EACH Order block	Explicit ON ERROR UNDO, RETURN

When the AVM raises ERROR in the FOR EACH ORDER block, the explicit ON ERROR phrase directs the AVM to return to the caller which is the procedure file. Since the RETURN option does not include the ERROR option, ERROR is not raised in the procedure block, and the final DISPLAY statement executes. However, the first DISPLAY statement (“For blocks complete”) does not run.

If you change the explicit ON ERROR phrase as shown in the following code snippet, you see almost identical behavior, except the final display statement does not execute:

```
FOR EACH Order OF Customer
    ON ERROR UNDO RETURN ERROR:
```

Due to the ON phrase shown, ERROR is then raised in the procedure block. The AVM then executes the default LEAVE action and return control to its caller. If this is the top-level procedure, the application ends. If you change the explicit ON ERROR phrase as shown in the following code snippet, an error object is created and raised in the outer block, which is the FOR EACH Customer block:

```
FOR EACH Order OF Customer
    ON ERROR UNDO THROW:
```

Since there is no explicit ON phrase, the default action occurs, which is that the error message is displayed and the AVM goes to the next Customer iteration.

If you remove the explicit ON ERROR phrase altogether, the implicit ON ERROR phrase is ON ERROR UNDO, NEXT, and one error message is displayed for each Order of each Customer record.

Using UNDO, THROW

You can use ON ERROR UNDO, THROW to change the default error handling. This construct is beneficial since it is not possible to use an ON phrase on some block types, such as procedure or method blocks. It is also useful for handling conditions in a central location, rather than locally. For example, if you have a code module with many blocks that can fail, and there is no advantage to a local CATCH block, and the error handling code is the same for all blocks in the module, then you can THROW all the errors up the call stack to a central location where a single CATCH statement can handle them all.

To take advantage of this more modern, structured error handling approach, you can change the default error directive to UNDO, THROW. Then errors propagate up by default so they can be handled by a common CATCH block. Exceptions can then be coded on specific blocks. To accomplish this, ABL provides two statements:

- BLOCK-LEVEL ON ERROR UNDO, THROW
- ROUTINE-LEVEL ON ERROR UNDO, THROW

BLOCK-LEVEL ON ERROR UNDO, THROW statement

The BLOCK-LEVEL ON ERROR UNDO, THROW statement changes the default implicit ON ERROR phrase to ON ERROR UNDO, THROW for every supported block type in the file that contains the statement. This is specifically for ERROR, not STOP conditions because STOP conditions are already thrown by default. The following blocks are affected by this statement:

- Procedure (also called main block, external procedure, or .p file)
- Internal procedure
- Database trigger (ON block with CREATE, DELETE, WRITE, or ASSIGN event)
- User-defined function
- Constructor
- User-defined method
- User-defined property getter/setter
- REPEAT
- FOR
- DO TRANSACTION

The following blocks are **not** affected:

- Any block for which an error-handling directive is explicitly specified
- Simple DO block
- DO WHILE block
- Destructor

- UI trigger

Syntax

```
BLOCK-LEVEL ON ERROR UNDO, THROW.
```

The following rules affect the placement of the `BLOCK-LEVEL ON ERROR UNDO, THROW` statement:

- The statement occurs once in each source file in which the behavior is desired.
- The statement must come before any definitional or executable statement in the procedure or class file.
- The statement can come before or after a `USING` statement.

Example

To create an application that uses structured error handling to handle all uncaught local errors at the top level:

1. Include the `BLOCK-LEVEL ON ERROR UNDO, THROW` statement in all your procedure and class files.
2. For each basic block, decide whether a different explicit flow of control directive is appropriate.
3. Add a `CATCH` block for the `Progress.Lang.Error` interface to your startup procedure block. For more information, see CATCH Blocks on page 65.
4. Add a `CATCH` block locally for any errors you want to handle at a local level.

The following simple example illustrates the design pattern:

```
BLOCK-LEVEL ON ERROR UNDO, THROW.

PROCEDURE find1000:
    /* Ignore potential errors */
    FIND FIRST Customer WHERE CustNum = 1000 NO-ERROR.
END PROCEDURE.

PROCEDURE find2000:
    FIND FIRST Customer WHERE CustNum = 2000.

    CATCH eSysError AS Progress.Lang.SysError:
        /* Take care of this error locally */
    END CATCH.
END PROCEDURE.

PROCEDURE find3000:
    FIND FIRST Customer WHERE CustNum = 3000.
END PROCEDURE.

/* Main Startup Procedure Block */

RUN find1000.
RUN find2000.
RUN find3000.

/* Won't execute because error will be raised here by find3000 */
MESSAGE "Application completed execution successfully."
    VIEW-AS ALERT-BOX BUTTONS OK.

CATCH eAnyError AS Progress.Lang.Error:
    MESSAGE "Unexpected error occurred..." SKIP
            "Logging information..." SKIP
            "Exiting application..."
                VIEW-AS ALERT-BOX BUTTONS OK.
END CATCH.
```

ROUTINE-LEVEL ON ERROR UNDO, THROW statement

The ROUTINE-LEVEL ON ERROR UNDO, THROW statement is very similar to BLOCK-LEVEL ON ERROR UNDO, THROW, but it affects only a subset of block types. Specifically, this statement changes the default behavior for the following blocks:

- Procedure (also called main block, external procedure, or .p file)
- Internal procedure
- Database trigger (ON block with CREATE, DELETE, WRITE, or ASSIGN event)
- User-defined function
- Constructor
- User-defined method
- User-defined property getter/setter

The following blocks are **not** affected:

- Any block for which an error-handling directive is explicitly specified
- REPEAT
- FOR
- DO TRANSACTION
- Simple DO block
- DO WHILE block
- Destructor
- UI trigger

Syntax

```
ROUTINE-LEVEL ON ERROR UNDO, THROW.
```

The same rules that affect the placement of the BLOCK-LEVEL ON ERROR UNDO, THROW statement also apply to ROUTINE-LEVEL ON ERROR UNDO, THROW:

- The statement occurs once in each .p or .cls file in which the behavior is desired.
- The statement must come before any definitional or executable statement in the procedure or class file.
- The statement can come before or after a USING statement.

Note: The ROUTINE-LEVEL ON ERROR UNDO, THROW statement is ignored if BLOCK-LEVEL ON ERROR UNDO, THROW occurs in the same file.

-undothrow startup parameter

The compile-time startup parameter `-undothrow n` makes `UNDO, THROW` the default block-level or routine-level error directive for all files compiled while the parameter is in effect. It has the same effect as inserting either `BLOCK-LEVEL ON ERROR UNDO, THROW` or `ROUTINE-LEVEL ON ERROR UNDO, THROW` (depending on the value of n) in every source file compiled into r-code.

Caution: Because this parameter potentially affects many files comprising a large volume of source code, be sure that you understand the implications of using it to compile your application.

The argument *n* is required and must have a value of 1 or 2:

- `-undothrow 1` — Yields the same result as including `ROUTINE-LEVEL ON ERROR UNDO, THROW` in every procedure and class file being compiled. See ROUTINE-LEVEL ON ERROR UNDO, THROW statement on page 40.
- `-undothrow 2` — Yields the same result as including `BLOCK-LEVEL ON ERROR UNDO, THROW` in every procedure and class file being compiled. If any `ROUTINE-LEVEL ON ERROR UNDO, THROW` statements occur in the source, this parameter supersedes them. See BLOCK-LEVEL ON ERROR UNDO, THROW statement on page 38.

Using this parameter with an argument other than 1 or 2 results in an error and immediate termination of the process.

Determining error-handling characteristics of r-code

To determine whether a given r-code file was compiled with either `BLOCK-LEVEL ON ERROR UNDO, THROW` or `ROUTINE-LEVEL ON ERROR UNDO, THROW` in effect (applied by using either a statement or the `-undothrow` parameter), inspect one of the following:

- The `UNDO-THROW-SCOPE` attribute of the `RCODE-INFO` system handle
- The `COMPILE` statement's `XREF` or `XREF-XML` output

UNDO-THROW-SCOPE attribute

The `RCODE-INFO` system handle has a read-only attribute named `UNDO-THROW-SCOPE` of type `CHARACTER`. It has the following possible values:

- "`ROUTINE-LEVEL`"
- "`BLOCK-LEVEL`"
- "" (empty string), if neither directive is used

XREF and XREF-XML output

If a file compiled with the `XREF` option is subject to block-level or routine-level `UNDO, THROW` behavior, the `XREF` output includes one of the following lines:

```
<compile file_name> <file_name> <line #> ROUTINE-LEVEL ON ERROR UNDO, THROW
<compile file_name> <file_name> <line #> BLOCK-LEVEL ON ERROR UNDO, THROW
```

The line, if present, is usually the first line in the listing for the applicable file, and in all cases is near the top of the listing.

Similarly, if the file is compiled with the XREF-XML option, the XML output includes an entry like the following ("BLOCK-LEVEL" replaces "ROUTINE-LEVEL" as appropriate):

```
<Reference Reference-type="ROUTINE-LEVEL" Object-identifier="">
      <Source-guid>xX6qZFj+b6fhERbFDfGViA</Source-guid>
      <File-num>1</File-num>
      <Ref-seq>3</Ref-seq>
      <Line-num>1</Line-num>
      <Object-context>ON ERROR UNDO, THROW</Object-context>
      <Access-mode/>
      <Data-member-ref/>
      <Temp-ref/>
      <Detail/>
      <Is-static>false</Is-static>
      <Is-abstract>false</Is-abstract>
</Reference>
```

Note: Where neither the block-level nor the routine-level directive is in effect, no explicit corresponding entry appears in the output. If the source file contains both statements, both corresponding entries appear in the output, even though the AVM ignores the routine-level statement.

ON phrases and STOP conditions

There are a few STOP conditions that do not abide by some or all ON phrases. Here are two prominent ones:

- The ABL code tries to use an inactive index.

  The AVM propagates the STOP condition up through the block that started the transaction (not just the subtransaction) regardless of any ON STOP phrases or CATCH blocks along the way.

- The database connection is lost.

  When the connection is lost, any code that references the database causes subsequent errors to occur. The AVM attempts to avoid these cascading error conditions by raising the STOP condition and propagating it up to the first procedure or class level above which there are no references to the database. Any ON STOP phrases or CATCH blocks encountered along the way are ignored.

Throwing error and stop objects from an AppServer to an ABL client

If an error is thrown out of a top level procedure of an AppServer (for example, by using RETURN ERROR *error-object-expression* or UNDO, THROW *error-object-expression*), the error or stop object being thrown is serialized and sent back to the ABL client. The client then deserializes the object and rethrows it in the context of the RUN statement on the client. This functionality is subject to the same serialization/deserialization restrictions as for any other object. The restrictions particularly relevant to error and stop objects are as follows:

- In the case of a user-defined class, the object's class and all the classes in its hierarchy must be marked as SERIALIZABLE. For more information on marking a class SERIALIZABLE, see the CLASS statement in the *ABL Reference*.

- .NET and ABL-extended .NET error objects cannot be thrown across the AppServer boundary.
- `SoapFaultError` objects can be thrown from an AppServer to an ABL client. However, the handle-based object that the `SoapFault` property points to is not recreated during the deserialization of the `SoapFaultError` object. It is set to the Unknown (`?`) value.

In the case of the first two items, if the AppServer code attempts to throw such an object, any message from the object is written to the AppServer log. In addition, another error is raised to indicate that the throw failed. That error message is also written to the AppServer log. An error condition is raised on the `RUN` statement in the client.

Class-based error and stop objects can also be thrown from an OpenEdge application server to a client for an asynchronous request. In that case, error and stop conditions will not be handled by a `CATCH` block as the block containing the `RUN` statement may be long over. Instead, the information must be made available in the `PROCEDURE-COMPLETE` event handler via attributes of the asynchronous request handle. Therefore, an error object or `Progress.Lang.StopError` stop object is returned to the client and its reference provided as the value of the `ERROR-OBJECT` attribute of the asynchronous request handle. Any other stop object (a `Progress.Lang.Stop` or a subclass) is returned to the client and its reference provided as the value of the `STOP-OBJECT` attribute of the asynchronous request handle. The `ERROR-STATUS` system handle's `ERROR` attribute is also set.

5

ERROR and STOP Classes

There is a set of built-in objects in ABL that represent system-generated `ERROR` conditions:

- `Progress.Lang.ProError`
- `Progress.Lang.SysError`
- `Progress.Lang.SoapFaultError`

There is also a set of built-in objects that represent system-generated `STOP` conditions:

- `Progress.Lang.StopError`
- `Progress.Lang.Stop`
- `Progress.Lang.StopAfter`
- `Progress.Lang.LockConflict`
- `Progress.Lang.UserInterrupt`

There is also a built-in object which can be used to represent an application-generated error condition. In addition, you can create your own application error objects that inherit from this class:

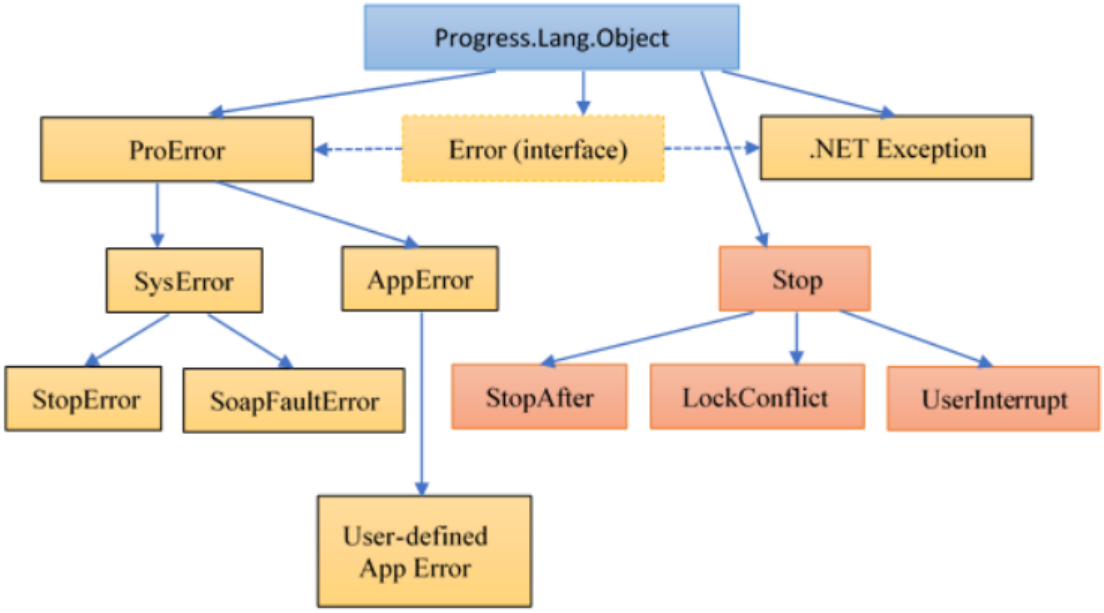
- `Progress.Lang.AppObject`

Lastly, there is an interface that contains the common properties of all error objects as well as the `StopError` object:

- `Progress.Lang.Error`

The following diagram shows the relationship between these objects.

Figure 1: Hierarchy of Error and Stop Classes



Since `Progress.Lang.ProError` implements the `Progress.Lang.Error` interface, all of its subclasses do as well. Note that a .NET Exception also behaves as if it implements the `Progress.Lang.Error` interface even though it is not an ABL object. Other than `Progress.Lang.StopError`, the other stop objects do not implement this interface.

For details, see the following topics:

- Progress.Lang.Error interface
- Progress.Lang.ProError class
- Progress.Lang.SysError class
- Progress.Lang.StopError class
- Progress.Lang.SoapFaultError class
- Progress.Lang.Stop class
- Progress.Lang.StopAfter class
- Progress.Lang.UserInterrupt class
- Progress.Lang.LockConflict class
- Progress.Lang.AppError class
- .NET Exceptions
- Enabling stack tracing with error objects

Progress.Lang.Error interface

The `Progress.Lang.Error` interface describes a common set of properties and methods that built-in ABL error classes implement. This interface cannot be implemented by a user-defined class. For user-defined classes, create a subclass of the `Progress.Lang.AppError` class to create your own type of ABL error object. See the Progress.Lang.AppError class on page 50 for more information.

The `Progress.Lang.Error` interface defines the properties and methods shown in the following table:

Table 3: Properties and methods

Member	Description
`CallStack` property	Returns a string representing the call stack at the time the error object was created. For `Progress.Lang.StopError`, this property is always populated. For all other error objects, if the `ERROR-STACK-TRACE` attribute of the `SESSION` handle is false, then this property returns the Unknown value (`?`). To enable the call stack, set `SESSION:ERROR-STACK-TRACE` property to `TRUE` directly, or use the `-errorstack` startup parameter.
`NumMessages` property	This property indicates how many error messages the error object contains.
`Severity` property	The `Severity` property is not used by ABL system errors and returns zero if accessed. It is provided as a mechanism to assign severity rankings to your various application errors (`Progress.Lang.AppError`).
`GetMessage(` *`MessageIndex`* `)` method	Returns the error message for the error at the specified index position in the object's message list,beginning with one (1). If there is no error message at the indicated index, the method returns the empty string.
`GetMessageNum(` *`MessageIndex`* `)` method	Returns the error message number for the error at the specified index position in the object's message list. For the `Progress.Lang.SysError` and `Progress.Lang.SoapFaultError` objects, the method returns the unique message number for the system generated error. If there is no error message at the index, the method returns 0. If the object is a .NET Exception, the method also returns 0.

Progress.Lang.ProError class

`Progress.Lang.ProError` is the super class for all ABL built-in and user-defined classes that represent errors in the ABL. You cannot directly inherit from this class, and the class constructors are reserved for system use only. 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

`Progress.Lang.ProError` inherits from `Progress.Lang.Object` and therefore inherits all the common methods and properties needed for managing user-defined objects in ABL. It also implements the `Progress.Lang.Error` interface, which provides all the properties and methods relevant for an error object, as shown in Properties and methods table.

Progress.Lang.SysError class

When an ABL statement generates an error message and raises the `ERROR` condition, the AVM creates a `Progress.Lang.SysError` object. You cannot inherit from this class, and the class constructors are reserved for system use only.

The Properties and methods table describes the properties and methods implemented by this class.

Progress.Lang.StopError class

When an ABL statement generates an error message that raises the `STOP` condition, the AVM creates a `Progress.Lang.StopError` object. You cannot inherit from this class, and the class constructors are reserved for system use only.

The Properties and methods table describes the properties and methods implemented by this class.

Progress.Lang.SoapFaultError class

This class wraps the ABL built-in SOAP-fault system object. The SOAP-fault object contains the information from a SOAP fault generated by a Web service call from an ABL application. `Progress.Lang.SoapFaultError` inherits from `Progress.Lang.SysError`. You cannot inherit from this class, and the class constructors are reserved for system use only.

The Properties and methods table describes the properties and methods implemented by this class. The following table describes the additional property of this class.

Table 4: SoapFaultError properties

Member	Description
`SoapFault` property	Contains the handle to the SOAP-fault object. The `Soap-Fault-Detail` property of this handle provides the full detail about the original SOAP fault, among other attributes.

See the section on handling errors in *Develop Web Services for OpenEdge* for more detailed information on handling SOAP faults.

Caution: Like other error objects, objects of type `SoapFaultError` can be thrown from an AppServer to an ABL client. However, the handle-based object that the `SoapFault` property points to is not recreated during the deserialization of the `SoapFaultError` object.

Progress.Lang.Stop class

When the AVM executes the `STOP` statement it creates an instance of the `Progress.Lang.Stop` class. You cannot inherit from this class, and the class constructors are reserved for system use only. It contains one property as shown in the table below.

Table 5: Progress.Lang.Stop properties

Member	Description
`CallStack` property	Returns a string representing the call stack at the time the stop object is created. Because this is for a `STOP` condition, this property is always populated. The `ERROR-STACK-TRACE` attribute of the `SESSION` handle does not have to be `TRUE`.

Progress.Lang.StopAfter class

This object is created when there is a timeout due to a `STOP-AFTER` phrase. This class inherits from `Progress.Lang.Stop` and thus inherits the `CallStack` property. It has no methods or properties of its own.

Progress.Lang.UserInterrupt class

This object is created when the user hits **CTRL+C** (Unix/Linux) or **CTRL+Break** (Windows). This class inherits from `Progress.Lang.Stop` and thus inherits the `CallStack` property. It has no methods or properties of its own.

Progress.Lang.LockConflict class

This object is created when there is a timeout while waiting for a record lock (based on the `-lkwtmo` startup parameter), or by hitting **Cancel** on the lock conflict wait dialog. Note that in character mode (Linux, Unix or character mode in Windows), you can hit **CTRL+C** or **CTRL+Break** to stop waiting. However, the AVM still maps that to the `LockConflict` object due to the context, not to the `Progress.Lang.UserInterrupt` object.This class inherits from `Progress.Lang.Stop` and thus inherits the `CallStack` property. It also has three properties of its own. These all correspond to the same information that is shown in the lock conflict wait dialog box.

Table 6: Progress.Lang.LockConflict properties

Member	Description
`TableName` property	This is the name of the database table that has the lock conflict.
`User` property	This is the name of the user that is currently holding the lock and thus causing the lock conflict.
`Device` property	This is the name of the device on which the other AVM process is running that is holding the lock and thus causing the lock conflict. Alternatively, this can be “Dictionary” if another process is doing schema updates via the Dictionary. The format of this name is different on different operating systems.

Progress.Lang.AppError class

`Progress.Lang.AppError` is the 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 an error in the ABL structured error handling model. An application can use the built-in `AppError` class directly or can create objects that inherit from this class to provide extra error or contextual information

An application error can be raised either by using the `RETURN ERROR` or the `UNDO, THROW` statement. See Raising Conditions on page 55 for details on how to throw an application error.

The following table describes additional properties and methods of this class beyond what is required by the `Progress.Lang.Error` interface, which this class implements.

Table 7: AppError properties and methods

Member	Description
`ReturnValue` property	This property is included in the `AppError` object to provide a bridge between the older functionality of `RETURN ERROR` *`ErrorString`* and error objects. Traditionally, this form of the `RETURN ERROR` statement populated the data for the `RETURN-VALUE` function. Now, the AVM also generates an `AppError` and populates the `ReturnValue` property. That way the same information is available if this error object is caught. Without using `CATCH` blocks, the `ErrorString` is still available in the traditional way, via the `RETURN-VALUE` function.
`Severity` property	Although the `Severity` property is an inherited property, it is intended as a feature of `AppError` objects and is not used by `SysError` objects. `Severity` has no intrinsic meaning to ABL. You can use it to establish a severity ranking system in your application.
`AddMessage(`*`ErrorMessage, MessageNumber`*`)` method	Adds a message to the `AppError` object with values from the *`ErrorMessage`* and *`MessageNumber`* arguments to the end of the message list. Your application provides the message number and text. Access error messages and message numbers with the `GetMessage()` and `GetMessageNum()` methods.This method increments the `NumMessages` property on the `AppError` by 1.
`RemoveMessage(`*`MessageIndex`*`)` method	Removes the error at the specified index position (both error message string and error message number) from the message list. The method decrements the `NumMessages` property by 1 and moves the messages after the indexed error forward in the list by 1.

AppError Constructors

The following is the default constructor. This constructor creates an `AppError` object with an empty message list and does not set any properties.

Syntax

```
PUBLIC AppError( )
```

The following constructor creates an `AppError` object and assigns the first message on the object with the values from the *`ErrorMessage`* and *`MessageNumber`* arguments. It also sets the `NumMessages` property to 1. The error message and message number can be accessed with the `GetMessage(1)` and `GetMessageNum(1)` methods.

Syntax

```
PUBLIC AppError( INPUT ErrorMessage   AS CHARACTER
                 INPUT MessageNumber AS INTEGER )
```

The following constructor creates an `AppError` object with the `ReturnValue` property set with the value of the `ReturnValue` parameter. This constructor is used when the AVM implicitly creates an `AppError` object for a `RETURN ERROR ErrorString` statement. You can also invoke this constructor directly.

Syntax

```
PUBLIC AppError( INPUT ReturnValue AS CHARACTER )
```

Note: This constructor does not set an error message. If the `AddMessage()` method is not used to set one, no message is displayed if an object constructed this way is thrown and handled by default error handling.

Below is an example of using an `AppError`.

```
ROUTINE-LEVEL ON ERROR UNDO, THROW.
DEF VAR ix AS INT.

RUN proc.

CATCH err AS PROGRESS.lang.AppERROR:

   MESSAGE "An Error occurred" SKIP
           "returnvalue" err:returnvalue SKIP // This will be empty
           "severity" err:severity SKIP
           VIEW-AS ALERT-BOX.

   DO ix = 1 TO err:NUMMESSAGES: // There will be 2
       MESSAGE err:GetMessage(ix) err:GetMessageNum(ix).
   END.
END.

PROCEDURE proc:
    DEFINE VAR err AS PROGRESS.Lang.Apperror.

    err = NEW PROGRESS.Lang.AppError("The car cannot be rented", 1).
    err:addmessage ("No driver's license was provided", 25).
    err:severity = 10.

    /* This is thrown to the caller due to the ROUTINE-LEVEL
       ON ERROR UNDO, THROW directive. */
    UNDO, THROW err.

END.
```

.NET Exceptions

A .NET Exception is a class instance that is thrown by a .NET method that inherits from .NET's `System.Exception`. If you are interacting with .NET objects through your ABL code, it is possible that one of these error objects can be thrown. In ABL you interact with these .NET objects in the same way as with ABL error and stop objects. They can be caught or rethrown.

The .NET native methods and properties of the Exception object can be accessed. However, ABL also makes .NET exceptions appear as if they implement the `Progress.Lang.Error` interface. So you can pass a .NET exception to an error handling routine that accepts a `Progress.Lang.Error` interface as a parameter. Then you can use the methods and properties of that interface to access the data in the .NET class.

Enabling stack tracing with error objects

All error objects have the ability to preserve the call stack in the `CallStack` property. The property is populated at the time an error object is instantiated. Populating the `CallStack` property incurs a small amount of overhead that you may not want. Therefore, ABL has an attribute on the `SESSION` handle called `ERROR-STACK-TRACE` and a startup parameter called `-errorstack` to enable or disable this feature. The default value is `FALSE` (disabled). However, if getting the call stack is important for troubleshooting your application, you should not hesitate to use it. Note that it is not necessary to use this to get call stack information from any of the Stop classes (`Progress.Lang.StopError`, `Progress.Lang.Stop`, or any of its subclasses). For `STOP` conditions, `CallStack` is always populated.

If `ERROR-STACK-TRACE` is `FALSE`, the `CallStack` property of an error object is the Unknown value (`?`).

If `ERROR-STACK-TRACE` is `TRUE`, an error object thrown from an AppServer to a client contains the stack trace from the AppServer in its `CallStack` property. If `ERROR-STACK-TRACE` is `TRUE` on the client, the `CallStack` property also includes the client stack trace. The text `"Server StackTrace:"` appears at the top of the AppServer stack trace to differentiate it from the client stack trace.

```
doIt asclient.p at line 46  (asclient.p)
asclient.p at line 26  (asclient.p)
main.p at line 10  (main.p)

Server StackTrace:
myproc svr.p at line 30 (svr.p)
svr.p at line 5  (svr.p)
```

6

Raising Conditions

When a system condition is encountered during program execution, the AVM automatically raises an `ERROR` or `STOP` condition. However, there are also programmatic ways to raise a condition. This section discusses ABL constructs for raising conditions programmatically.

For details, see the following topics:

- Raising errors with UNDO, THROW
- RETURN ERROR
- Raising ERROR to the caller of a user-defined function
- Throwing a condition out of a destructor
- Raising the QUIT condition
- Raising the STOP condition
- Raising a timed STOP condition
- Throwing error and stop objects from an AppServer to an ABL client

Raising errors with UNDO, THROW

During the execution flow, you may determine that there is an invalid condition (for example, when validating input) and want to communicate that to other parts of the program or to the user. To do this, you create an application error object and raise the `ERROR` condition by using the `UNDO, THROW` statement.

In addition, you can provide local CATCH blocks that provide handling for specific error numbers and throw any other condition to an outer or calling block to be handled by more generic error-handling code. The same UNDO, THROW statement can be used in a CATCH block to accomplish this.

Syntax

```
UNDO, THROW [error-object-expression] .
```

error-object-expression

Any expression that results in an instance of an error or stop object.

When this statement is executed, it undoes the current block (the innermost block with error-handling capabilities) and raises ERROR or STOP on the statement. ERROR is raised if it is an error object and STOP is raised for any of the stop objects (which includes Progress.Lang.StopError). It is then up to the block directives to determine how the error is handled, as with any other condition.

Examples

In each of the following examples, ERROR is raised on the UNDO, THROW statement. If there are any updates in the block that require undoing, that is performed, though in these examples there are not. The comment in each code example describes what happens. Assume the current output device is the screen:

Example 1

This example demonstrates the use of UNDO, THROW within a block. Note that this is not the same as using the ON ERROR UNDO, THROW directive on that block. With the UNDO, THROW statement, the condition is simply raised on the current statement.

```
DO ON ERROR UNDO, LEAVE:
    IF CurrentTime > ClosingTime THEN
       UNDO, THROW NEW Progress.Lang.AppError("Can't take a delivery order
                                              after closing time.", 550).
END.

<next line>

/* Since there is no CATCH block and no THROW directive, the error message
   provided as the parameter to the AppError constructor is displayed.
   Execution then continues at <next line>.
*/
```

Example 2

With the ON ERROR UNDO, THROW directive, the condition, which was already raised within the context of that block, is thrown to the outer block (or caller if there is no outer block).

```
DO ON ERROR UNDO, THROW:
   IF CurrentTime > ClosingTime THEN
       UNDO, THROW NEW Progress.Lang.AppError("Can't take a delivery order
                                              after closing time.", 550).
END.

<next line>

/* Once error is raised, the newly created AppError object is thrown out of the DO
   block due to the ON ERROR UNDO, THROW block directive. Thus, error is raised again
   at the END of the DO block (though functionally, it is outside of this block).
   Because there is no THROW directive or CATCH block at this level, this causes
   <next line> not to execute.  Instead the error message that was provided as the
   parameter to the AppError constructor is displayed. Assuming this is the end
   of a procedure, execution continues in the caller, if there is one.
*/
```

If you use the DEBUG-ALERT feature (SESSION:DEBUG-ALERT = yes, or startup parameter -debugalert) for each of the above two examples, the difference is clearer. For the first example, if you hit the **Help** button on the alert box when the error message is displayed, it shows line 3 (the UNDO, THROW statement). In the second example, it shows line 5 (the END statement).

Example 3

This example demonstrates what happens when the UNDO, THROW statement occurs within a CATCH block. Here the condition is raised and thrown to the outer block of the associated block (or the caller if there is no outer block). This happens because CATCH blocks have the UNDO, THROW flow-of-control directive by default.

```
DO ON ERROR UNDO, LEAVE:
    FIND FIRST Customer WHERE Customer.Name Begins "A".

    CATCH err AS Progress.Lang.Error:
        IF err:GetMessageNum(1) = 565 // record not found
        THEN <do custom handling for this error>
        ELSE UNDO, THROW err.
    END.
END.

<next line>

CATCH err AS Progress.Lang.Error:
    MESSAGE err:GetMessage(1)
        VIEW-AS ALERT-BOX.
END.

/* The caught Progress.Lang.SysError object is thrown out of the CATCH block
   and thus, out of the associated DO block.  Error is raised again in the
   outer block, <next line> does not execute, and the outer CATCH block runs.
*/
```

RETURN ERROR

The RETURN ERROR statement is another way of raising an ERROR condition. The full syntax of the RETURN statement is shown below, though here we are only discussing the RETURN ERROR options.

You can use the ERROR option in a procedure, database trigger block, class-based method, constructor, or property accessor. However, you cannot use it in a user-interface trigger block or destructor to raise ERROR outside of that block. This results in a compiler error. More information about conditions in destructors are discussed in Throwing a condition out of a destructor on page 60. You also cannot use RETURN ERROR in a user-defined function to raise ERROR outside of that block. This is explained in more detail in Raising ERROR to the caller of a user-defined function on page 59.

Note that if an error is returned from any procedure, method, or user-defined function, the values of any OUTPUT or INPUT-OUTPUT parameters are not returned to the caller.

Syntax

```
RETURN
    [ return-value |
    ERROR [ error-return-value | error-object-expression ] |
    NO-APPLY ] .
```

error-return-value

A character expression.

error-object-expression

Any expression that results in an instance of an error or stop object.

Aside from being a stand-alone statement, the RETURN phrase, with the same options, is also available on the following language elements:

- ON ENDKEY phrase
- ON ERROR phrase
- ON QUIT phrase
- ON STOP phrase
- UNDO statement

Regardless of the options, when RETURN ERROR executes, execution returns to the caller of the current procedure, method, constructor, or property accessor, and ERROR is raised in the caller. If it occurs in a database trigger, ERROR is raised on the statement that caused the database event. Because the AVM returns before raising error, the block containing the RETURN statement or phrase is not undone. The behavior in the caller is then dictated by that block's error-handling capabilities (error directive or CATCH block).

Note: The remainder of this section does not apply to user-defined functions.

The AVM does the following when the RETURN ERROR statement executes:

- If neither an *error-return-value* nor an *error-object-expression* is supplied, the AVM creates an instance of Progress.Lang.AppError with no error message set and the ReturnValue property set to the empty string. The value for the RETURN-VALUE built-in function is also set to the empty string.
- If an *error-return-value* expression is supplied, the AVM creates the AppError object and sets its ReturnValue property. No error message is set in the object. It also sets the value for the RETURN-VALUE built-in function.

- If an instance of an error object is supplied, the AVM does not create a new instance.
- In all cases, the AVM returns and behaves as if this object was thrown on the invoking statement.

In the caller, if NO-ERROR is used on the invoking statement, any error messages in the object are transferred to the ERROR-STATUS system handle, ERROR-STATUS:ERROR is set to TRUE, and the error object instance is discarded. Any custom information in the error object is lost.

Otherwise (if NO-ERROR is **not** used on the invoking statement) the error object is handled in the usual way. If there is a relevant CATCH block, the error is caught. Otherwise the block's error directive takes effect. If default error handling is in place, any error message is displayed to the output device. But if there is no error message stored in the object, no display is made.

Example

The following example shows how RETURN ERROR is used to return a custom AppError to a caller using NO-ERROR:

```
DEF VAR ix AS INT.

RUN proc NO-ERROR.
IF ERROR-STATUS:ERROR THEN
DO:
    DO ix = 1 TO ERROR-STATUS:NUM-MESSAGES: /* this will be 2 */
        MESSAGE ERROR-STATUS:GET-MESSAGE(ix) ERROR-STATUS:GET-NUMBER(ix).
    END.
END.

PROCEDURE proc:
    DEFINE VAR err AS PROGRESS.Lang.AppError.

    err = NEW PROGRESS.Lang.AppError("The car cannot be rented",1).
    err:AddMessage ("No driver's license was provided", 98).

    RETURN ERROR err.
END.
```

Raising ERROR to the caller of a user-defined function

The user-defined function, defined by the FUNCTION statement, returns a value of a specific data type as its primary function. The RETURN statement is used in the function body to specify what value to return to the caller. The RETURN ERROR statement is not used in the same way as it is in other blocks since it does not raise ERROR in the caller. Instead, it sets the target variable of the function to the Unknown value (?). Therefore, you can perform error checking on a function call by checking for the Unknown value (?). This technique only works if the target variable has a value other than the Unknown value (?) before the function is called.

The following code demonstrates this behavior:

```
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.".
```

If you specify a string (RETURN ERROR <string>), the string is not seen as a RETURN-VALUE, but as the value being returned from the function. Therefore, if the function is defined to return a type other than CHARACTER, you get a compiler error (or a runtime error if the expression type is indeterminate at compile time). If the function is defined to return a CHARACTER, the code runs, but RETURN-VALUE is not set. The specified string is lost.

Structured error handling provides a more consistent and robust way to raise ERROR from user-defined functions using the UNDO, THROW statement rather than RETURN ERROR. There are two ways to do this:

- Syntactically, you cannot use an ON ERROR phrase on a FUNCTION definition. Therefore, use the ROUTINE-LEVEL ON ERROR UNDO, THROW (or BLOCK-LEVEL ON ERROR UNDO, THROW) statement to set that directive on the function. Then any unexpected error, or explicitly thrown application errors, can be thrown back to the caller. Just be aware that this affects all routines or blocks in the file.
- Add a CATCH block to the function. With a CATCH block, any error that is caught can be thrown out of the function using the UNDO, THROW statement from within the CATCH block. If you only want to throw application errors back to the caller but handle system errors locally, or vice versa, you can use multiple CATCH blocks to accomplish this. Using multiple CATCH blocks is shown in the example below. To learn more, see CATCH Blocks on page 65.

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

FUNCTION ErrorTest RETURNS INTEGER:
   IF CurrentTime > ClosingTime THEN
      UNDO, THROW NEW Progress.Lang.AppError("Can't take a delivery order
                                             after closing time.", 1).
   CATCH err AS Progress.Lang.AppError:
      UNDO, THROW err. // Let the caller know
   END.

   CATCH err AS Progress.Lang.SysError:
      // Unexpected error; handle it here
      MESSAGE err:GetMessage(1) VIEW-AS ALERT-BOX.
   END.

END FUNCTION.

iFuncReturn = ErrorTest() NO-ERROR.
IF ERROR-STATUS:ERROR THEN
   MESSAGE "Error message returned from function:" SKIP
           ERROR-STATUS:Get-Message(1).
```

Throwing a condition out of a destructor

It is not possible to throw either an ERROR or STOP condition out of a class destructor. The destructor of a class can run at unexpected times, at the end of any statement, or at an END block where variables go out of scope. Therefore, you would not be able to effectively write code to handle any condition that was thrown out of the destructor block. Because of this, the AVM forces the condition to be handled within the block itself. It can be handled in all the usual ways, such as using an ON phrase (implicit or explicit) or by using a CATCH block.

Additionally, note that for destructors:

- Using ROUTINE-LEVEL ON ERROR UNDO, THROW or BLOCK-LEVEL ON ERROR UNDO, THROW, does not affect the destructor block of a class.
- Using the UNDO, THROW statement in the CATCH or FINALLY block of a destructor results in a compiler error because UNDO, THROW, in this context, would throw the condition out of the destructor block, which is not allowed. You can, however, use UNDO, THROW in the body of the destructor itself. This raises the

condition within the context of the block and it can be handled either by default error handling or by a local CATCH block.

- Using RETURN ERROR in the body of a destructor generates a compiler error.
- Unhandled STOP conditions normally propagate up by default. However this does not occur if the STOP condition is initiated within the destructor block. Instead, any error message is displayed to the current output device and the condition is cleared.

Raising the QUIT condition

The QUIT statement raises the QUIT condition.

Syntax

```
QUIT.
```

When the QUIT condition occurs, the AVM performs these steps by default:

1. Commits the current transaction.
2. Exits the ABL session.
3. For a client, if the application was started from the Procedure Editor or Progress Developer Studio for OpenEdge, the AVM returns to that tool; otherwise it returns to the operating system. If running in an AppServer, the AVM returns to the client session that called it.

If there is an ON QUIT phrase on the current block, that directive overrides the default behavior.

Raising the STOP condition

The STOP statement allows you to raise the STOP condition.

Syntax

```
STOP.
```

The STOP condition can be handled by the following constructs, in this order of precedence:

- An appropriate CATCH block
- An ON STOP phrase
- Default stop handling

Raising a timed STOP condition

The STOP-AFTER phrase specifies a time-out value for a DO, FOR, or REPEAT block. This is the syntax in the context of a DO block :

Syntax

```
DO ON ERROR UNDO, LEAVE STOP-AFTER expression:
   <body of the DO block>
END.
```

The integer *expression* specifies the number of seconds each iteration of a block has until a time-out occurs. If a time-out occurs, the AVM raises the STOP condition.

This STOP condition can be handled like other STOP conditions (for example, by using an ON STOP phrase), or it can be specifically handled by using a CATCH block for a Progress.Lang.StopAfter object. Specifically handling the condition is the recommended approach since it is the only way to know that the STOP condition was indeed raised by the STOP-AFTER and not by some other unexpected circumstance occurring within the block.

For more information on using this feature, see the STOP-AFTER phrase on the DO, FOR, or REPEAT statements in the *ABL Reference*.

Throwing error and stop objects from an AppServer to an ABL client

If an error is thrown out of a top level procedure of an AppServer (for example, by using RETURN ERROR *error-object-expression* or UNDO, THROW *error-object-expression*), the error or stop object being thrown is serialized and sent back to the ABL client. The client then deserializes the object and rethrows it in the context of the RUN statement on the client. This functionality is subject to the same serialization/deserialization restrictions as for any other object. The restrictions particularly relevant to error and stop objects are as follows:

- In the case of a user-defined class, the object's class and all the classes in its hierarchy must be marked as SERIALIZABLE. For more information on marking a class SERIALIZABLE, see the CLASS statement in the *ABL Reference*.
- .NET and ABL-extended .NET error objects cannot be thrown across the AppServer boundary.
- SoapFaultError objects can be thrown from an AppServer to an ABL client. However, the handle-based object that the SoapFault property points to is not recreated during the deserialization of the SoapFaultError object. It is set to the Unknown (?) value.

In the case of the first two items, if the AppServer code attempts to throw such an object, any message from the object is written to the AppServer log. In addition, another error is raised to indicate that the throw failed. That error message is also written to the AppServer log. An error condition is raised on the RUN statement in the client.

Class-based error and stop objects can also be thrown from an OpenEdge application server to a client for an asynchronous request. In that case, error and stop conditions will not be handled by a `CATCH` block as the block containing the `RUN` statement may be long over. Instead, the information must be made available in the `PROCEDURE-COMPLETE` event handler via attributes of the asynchronous request handle. Therefore, an error object or `Progress.Lang.StopError` stop object is returned to the client and its reference provided as the value of the `ERROR-OBJECT` attribute of the asynchronous request handle. Any other stop object (a `Progress.Lang.Stop` or a subclass) is returned to the client and its reference provided as the value of the `STOP-OBJECT` attribute of the asynchronous request handle. The `ERROR-STATUS` system handle's `ERROR` attribute is also set.

7

CATCH Blocks

ABL provides `CATCH` blocks to enable you to trap an error or stop object and write code to handle the object. This section discusses CATCH block syntax, usage, error handling precedence, UNDO scope, and nested CATCH blocks.

For details, see the following topics:

- Introduction to CATCH blocks
- CATCH block syntax and usage
- Blocks that support CATCH blocks
- Precedence of CATCH blocks
- UNDO scope and relationship to a CATCH block
- CATCH blocks within CATCH blocks

Introduction to CATCH blocks

A `CATCH` block can be referred to as an *end block* because it defines end-of-block processing for the block that encloses it. End blocks are always part of another block called the *associated block*. End blocks must appear in the associated block after the last executable statement and before the `END` statement. The other type of end block is the `FINALLY` block that is discussed in a later section.

The CATCH statement defines the start of an end block that only executes if a condition is raised in its associated block, and the type of condition raised is the type specified in the CATCH statement (or a subtype of that type). For example:

```
DO TRANSACTION ON ERROR UNDO, THROW:

    FIND FIRST Customer WHERE CustNum=1000.
    RUN CreditCheck.p(Customer.CustNum).

    /* CATCH associated with DO TRANSACTION */
    CATCH eAppError AS Progress.Lang.AppError:
        MESSAGE "This customer is on Credit Hold.".
    END CATCH.
END.

/* CATCH associated with Procedure (Main) block */
CATCH eSysError AS Progress.Lang.SysError:
    MESSAGE "Customer record does not exist.".
END CATCH.
```

In this example:

- The THROW directive tells the AVM to propagate any **unhandled** errors to the procedure (main) block, since it is the enclosing block of the DO TRANSACTION block. Notice there is a CATCH block waiting to handle any Progress.Lang.AppError object that may be raised from the RUN statement. If a Progress.Lang.AppError object is raised, the CATCH block handles the error and it is not passed to the procedure block.
- When running the code, if the FIND statement fails, and there is no error handler present for this error type, it raises a Progress.Lang.SysError. Since Progress.Lang.SysError is not handled, the AVM throws the error up the call stack to the procedure block, due to the UNDO, THROW directive of the TRANSACTION block. The AVM finds a compatible CATCH block on the procedure block and then executes the code in the CATCH block.
- If you delete the CATCH block on the procedure block and run the example code, the AVM propagates the Progress.Lang.SysError object to the main block as before. Since you no longer have an appropriate error handler in the main block, the AVM now executes the default error handling behavior, which is to display the system error message to the default output device.

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 condition type in a block. A CATCH block also handles objects for its subtypes, so it is possible there can be more than one CATCH block that is compatible with a particular condition. In this case, the AVM executes the first CATCH block it encounters that is compatible. For this reason, CATCH blocks should be arranged from the most specific type to the most general. For example, if you had different error handling code for Progress.Lang.SysError objects and Progress.Lang.SoapFaultError objects, put the CATCH block for SoapFaultError objects first. Otherwise, since SoapFaultError objects are a subtype of SysError, a CATCH block for SysError that appears first would handle the SoapFaultError object.

CATCH block syntax and usage

Syntax

```
CATCH object-variable AS [ CLASS ] condition-class:
    .
    .
    .
END [ CATCH ] .
```

object-variable

The variable name that references the object caught by this block. Typically, you do not define the *object-variable* ahead of time with the DEFINE VARIABLE statement. The AVM recognizes a new variable name on the CATCH statement as a new *object-variable* definition within the current scope. Each CATCH in an associated block must have a unique *object-variable*. You can reuse an *object-variable* name in a different associated block, if its type is the same as the previous use. For all blocks with their own variable scope, such as object methods or internal procedures, a CATCH statement inside that context may reuse the same variable name as a CATCH statement outside of that context even if the type is different.

[CLASS] *condition-class*

Optionally, you can provide the CLASS keyword.

The code within a CATCH block only executes if a condition of type *condition-class* (or a subtype) is raised within the body of the associated block. When the condition is raised, if there is an active transaction for the associated block, the transaction is undone before the AVM begins executing the statements within the CATCH block. For more information, see the reference entries for the DEFINE VARIABLE statement and the TRANSACTION option in the DO statement in the *ABL Reference*.

Examples

Example 1

In the following example, the CATCH block handles any ABL system error:

```
DEFINE VARIABLE iCust AS INTEGER.

ASSIGN iCust = 5000.

FIND Customer WHERE CustNum = iCust. /* Will fail */

/* Won't execute because FIND fails */
MESSAGE "Customer found" VIEW-AS ALERT-BOX BUTTONS OK.

/* The associated block for this CATCH block is the main block of the .p */
CATCH eSysError AS Progress.Lang.SysError:
    MESSAGE "From CATCH block..." SKIP
            eSysError:GetMessage(1)
        VIEW-AS ALERT-BOX.
END CATCH.
```

Example 2

The following example illustrates reuse of the *object-variable* name:

```
DEFINE VARIABLE oneError AS CLASS Progress.Lang.SysError.
    /* This definition is not necessary. */

DO ON ERROR UNDO, LEAVE:
    FIND FIRST Customer WHERE CustNum = 5000.

    CATCH oneError AS Progress.Lang.SysError:
        MESSAGE oneError:GetMessage(1) VIEW-AS ALERT-BOX.
    END CATCH.

    CATCH twoError AS Progress.Lang.AppError:
        MESSAGE twoError:GetMessage(1) VIEW-AS ALERT-BOX.
    END CATCH.
END. /* FIRST DO */

DO ON ERROR UNDO, LEAVE:
    FIND FIRST Customer WHERE CustNum = 6000.

    /* You can reuse an error-variable from a different
       associated block as long as it's the same type. */
    CATCH oneError AS Progress.Lang.SysError:
        MESSAGE oneError:GetMessage(1) VIEW-AS ALERT-BOX.
    END CATCH.

    /* NOT LEGAL: oneError was already used for a SysError,
       so it cannot be reused for an AppError. */
    CATCH oneError AS Progress.Lang.AppError:
        MESSAGE oneError:GetMessage(1) VIEW-AS ALERT-BOX.
    END CATCH.
END. /* SECOND DO */

PROCEDURE foo:
    FIND FIRST Customer WHERE CustNum = 7000.

   /* This IS LEGAL because a new oneError variable will be
      defined within the scope of this subprocedure so its
      type does not have to match. */
    CATCH oneError AS Progress.Lang.AppError:
        MESSAGE oneError:GetMessage(1) VIEW-AS ALERT-BOX.
    END CATCH.
END.
```

Example 3

An associated block may have multiple CATCH blocks, each of which handles a different error class. If an error type satisfies multiple CATCH statements, the AVM executes the code in the first CATCH block that is compatible with the error type. It does not execute multiple CATCH blocks. Therefore, if multiple CATCH blocks are specified, the more specialized error classes should come first, as shown:

```
FOR EACH Customer:

    < Code body of the associated block >

    /* This CATCH specifies the most specialized user-defined error class.
       It will catch only myAppError error objects or objects derived from
       myAppError. */

    CATCH eMyAppError AS Acme.Error.myAppError:
        /*Handler code for Acme.Error.myAppError condition. */
    END CATCH.

    /* This CATCH will handle Progress.Lang.AppError or any user-defined
       application error type, except for eMyAppError which is handled
       by the preceding CATCH block. */

    CATCH eAppError AS Progress.Lang.AppError:
        /* Handler code for AppError condition. */
    END CATCH.

    /* This CATCH will handle any error raised by an ABL statement.
       Since it is not in the class hierarchy of AppError, this CATCH
       could come before or after the CATCH for AppError */

    CATCH eSysError AS Progress.Lang.SysError:
        /* Handler code for SysError condition. */
    END CATCH.

    /* This is compatible with any condition object that
       implements the Progress.Lang.Error interface. All the
       above classes qualify, as well as a StopError object which
       is a SysError.  So, in this context, this CATCH block will
       only run for a .NET Exception. */

    CATCH eError AS Progress.Lang.Error:
        /* Handler code for any error condition. */
    END CATCH.

END. /* Associated Block */
```

Example 4

The compiler issues a warning message if a block contains a CATCH block that is not reachable. The following code produces a warning, since the CATCH of eMyAppError can never be reached:

```
FOR EACH Customer:
    /* Code body of the associated block */

    /* This will catch all application errors */

    CATCH eAppError AS Progress.Lang.AppError:
        /* Handler code for AppError condition */
    END CATCH.

    /* The following CATCH block will never execute, because
       myAppError is a subtype of Progress.Lang.AppError */

    CATCH eMyAppError AS Acme.Error.myAppError:
        /* Handler code for myAppError condition */
```

```
    END CATCH.

END. /* Associated Block */
```

Blocks that support CATCH blocks

The use of CATCH blocks is supported for all blocks that have error handling capabilities. CATCH cannot be used in a simple DO or DO WHILE block (one with no options), since these do not have implicit error handling; the compiler will not allow it. DO blocks must have an explicit TRANSACTION, ON ERROR UNDO, or ON STOP directive, in order to have a CATCH block.

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. This is the syntax for an associated block using end blocks:

Syntax

```
associated-block:
  .
  .
  .
  [ CATCH
    .
    .
    .
  END [ CATCH ] . ] . . .
  [FINALLY
    .
    .
    .
  END [ FINALLY ] . ]
END. /* associated-block */
```

Precedence of CATCH blocks

In general, the AVM performs error handling using this precedence, from highest to lowest. The AVM only abides by one of these when a condition is raised:

- Statement NO-ERROR option
- CATCH block
- Block's ON phrase (explicit or implicit)

Using NO-ERROR on a statement prevents a CATCH block from running if the statement raises a condition.

Otherwise a CATCH block takes precedence over any flow of control directive on the block, for example, LEAVE or THROW. See sections on Default Condition Handling on page 21 and Block Flow of Control and Condition Directives on page 31, for more information on block condition directives.

If there are CATCH blocks, but none of them are compatible with the type of condition that occurs, then the ON phrase for the block takes effect. This could be an explicit or implicit (default) phrase for the block type, such as ON ERROR or ON STOP. It can be useful to have both an explicit ON phrase for the associated block and a CATCH on the same associated block. You might want to CATCH certain error types and handle them directly, and have all other condition types handled by the ON phrase of the associated block.

UNDO scope and relationship to a CATCH block

Because the CATCH block only executes when ERROR or STOP is raised in the associated block, any transaction within the associated block is already undone. In other words, changes made within the associated block to persistent data, undo variables, and undo temp-table fields have been discarded. In addition, buffers scoped to the associated block of the CATCH block are not available when the CATCH block executes. This is because either the buffer was undone and released, or committed and released. If a buffer referenced in a CATCH block is referenced outside of the associated block, then the scope of that buffer is the smallest enclosing block outside of the associated block that encompasses all references to the buffer. Therefore, these buffers are available to the CATCH block.

The CATCH block itself is an undoable block with implicit ON ERROR UNDO, THROW error handling. You cannot explicitly override the ON ERROR directive for a CATCH block.

A statement that raises ERROR or STOP within a CATCH block causes the following to occur, unless the condition is handled within the block:

1. UNDO the CATCH block if it contains a transaction.
2. LEAVE the associated block.
3. THROW the condition to the block enclosing the associated block, or to the caller if there is no outer block.

The following example demonstrates these availability rules:

```
/* Defines an undoable variable (the NO-UNDO option is not specified). */
DEFINE VARIABLE TargetCustNum AS INTEGER.

/* The last valid value before the beginning of the DO block */
ASSIGN TargetCustNum = 1.

/* This block is a transaction block because it makes updates to the database */
DO ON ERROR UNDO, LEAVE:

    /* This value is undone on ERROR. */
    ASSIGN TargetCustNum = 15.

    /* Find a Customer */
    FIND Customer WHERE Customer.CustNum = TargetCustNum.

    /* Here's where we update the database. */
    ASSIGN Customer.Name = Customer.NAME + " And Much More".

    /* Confirm change to persistent field. */
    MESSAGE "Customer Name changed to: " Customer.Name
        VIEW-AS ALERT-BOX BUTTONS OK.

    /* ERROR raised. Control passes to CATCH block. */
    FIND Order OF Customer WHERE OrderNum = 1234.

    /* Statement does not execute. */
    DISPLAY Customer.CustNum SKIP
            Customer.Name SKIP
            OrderNum SKIP
            OrderStatus
                VIEW-AS TEXT WITH FRAME b SIDE-LABELS.

    CATCH eSysError AS Progress.Lang.SysError:

        /* Confirm if Customer record is available in CATCH.
           In this case it is not available because the customer
           record is released from the buffer when the associated
           block is undone, since it was never referenced in a
           higher block. */
        IF AVAILABLE (Customer) THEN DO:
            MESSAGE "Customer record is still available."
                VIEW-AS ALERT-BOX BUTTONS OK.
        END.
        ELSE DO:
            MESSAGE "No Customer record is currently available."
                VIEW-AS ALERT-BOX BUTTONS OK.

            /* Re-Find the Customer. Cannot rely on value of TargetCustNum! */
            FIND Customer WHERE Customer.CustNum = 15.

            /* Confirm that change to database field was not committed
               and UNDO variable was rolled back to 1. */
            MESSAGE "TargetCustNum = " TargetCustNum SKIP
                    "Customer name is now: " Customer.Name
                VIEW-AS ALERT-BOX BUTTONS OK.
        END. /* ELSE */

     END CATCH.
END. /* DO */
```

CATCH blocks within CATCH blocks

A CATCH block within a CATCH block only handles errors raised within the CATCH block. To prevent infinite looping, any UNDO, THROW statement within the top-level CATCH block, or any CATCH block nested within it, immediately throws the error to the block that encloses the associated block of the top-level CATCH block. For example:

```
FOR EACH Customer:
    < FOR EACH code body >

    DO ON ERROR UNDO, LEAVE:
        < DO code body >

        CATCH eAppError AS Progress.Lang.AppError:
            < CATCH code body >

            CATCH eSysError AS Progress.Lang.SysError:
                UNDO, THROW eSysError. /* Will be handled by CATCH
                                          anyError on FOR EACH... */
            END CATCH.
        END CATCH.
    END. /* DO */

    CATCH anyError AS Progress.Lang.Error:
        /* Handler code for anyError condition */
    END CATCH.

END. /* FOR EACH */
```

In this example, notice the UNDO, THROW statement within the nested CATCH block. If we get here, the AVM passes control to the block enclosing the associated block and raises ERROR there. In this case, the DO block is the associated block and the FOR EACH is the block enclosing the DO block. The CATCH anyError block on the FOR EACH block then handles the error.

8

FINALLY Blocks

A FINALLY block can be referred to as an *end block* because it 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*. The other type of end block is the CATCH block which is discussed in CATCH Blocks on page 65.

The purpose of a FINALLY block is to hold cleanup code that must execute regardless of what else executed in the associated block. The FINALLY block may include code to delete dynamic objects, write to logs, close outputs, and other routine, but necessary, tasks. A FINALLY block runs on each iteration of a block, even if the iteration results in an ERROR or STOP condition.

For details, see the following topics:

- Introduction to FINALLY blocks
- FINALLY block syntax and usage
- UNDO scope and relationship to a FINALLY block
- Examples using FINALLY blocks
- FINALLY blocks and STOP-AFTER
- Conflicts between the associated and FINALLY blocks

Introduction to FINALLY blocks

The FINALLY statement creates an end block that executes once at the end of each iteration of its associated block, regardless of whether the associated block executed successfully or raised a condition.

The FINALLY block executes after:

- Successful execution of the associated block.
- Each successful iteration of an iterating associated block.
- ERROR or STOP is raised in the associated block regardless of whether a CATCH block or ON phrase handles the condition.

The FINALLY block does not execute if:

- A QUIT statement is in effect and it is not handled.

Since a FINALLY block executes after an invoked CATCH block, it can also be used to perform common post-CATCH cleanup tasks, rather than repeating common code in all the CATCH blocks present in the associated block.

FINALLY block syntax and usage

Here is the syntax for a FINALLY block:

```
FINALLY:
  .
  .
  .
END [ FINALLY ] .
```

All ABL blocks, other than a simple DO or DO WHILE block (one without TRANSACTION or an ON phrase), can have a FINALLY block.

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 and before the END statement. 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.

UNDO scope and relationship to a FINALLY block

The transaction of the associated block is either complete (success) or undone (failure) when FINALLY executes.

Buffers scoped to the associated block of the FINALLY block are not available when the FINALLY block executes. This is because either the buffer was undone and released, or committed and released.

If a buffer referenced in a FINALLY block is referenced outside of the associated block, then the scope of that buffer is the smallest enclosing block outside of the associated block that encompasses all references to the buffer. Therefore, these buffers are available to the FINALLY block.

The FINALLY block itself is an undoable block with implicit ON ERROR UNDO, THROW error handling. You cannot explicitly override the ON ERROR directive for a FINALLY block.

A statement that raises ERROR or STOP within a FINALLY block causes the following to occur, unless the condition is handled within the block:

1. UNDO the FINALLY block if it contains a transaction, which would be uncommon.
2. LEAVE the associated block.
3. THROW the condition to the block enclosing the associated block, or to the caller if there is no outer block.

Examples using FINALLY blocks

The examples that follow demonstrate common use cases for FINALLY blocks.

Example 1

In Example 1, the FINALLY block executes before any flow-of-control options (LEAVE, NEXT, RETRY, RETURN, or THROW) are executed for the associated block. For iterating blocks, the FINALLY block executes after each iteration of the block:

```
DO ON ERROR UNDO, LEAVE:
    FIND Customer WHERE CustNum = 1000. /* Raises ERROR and execution goes to
                                           FINALLY block before the LEAVE
                                           option executes */

    MESSAGE "This message never appears because of ERROR condition."
        VIEW-AS ALERT-BOX BUTTONS OK.

    FINALLY:
        MESSAGE "Inside FINALLY block."
            VIEW-AS ALERT-BOX BUTTONS OK.
    END FINALLY.  /* LEAVE DO block here */

END. /* DO */

MESSAGE "Out of DO block." VIEW-AS ALERT-BOX BUTTONS OK.
```

If you run this code, you see the following messages:

```
** Customer record not on file (138)
Inside FINALLY block.
Out of DO block.
```

Example 2

In Example 2, after ERROR is raised, execution goes to the CATCH block and then to the FINALLY block.

```
DO ON ERROR UNDO, LEAVE:
    FIND Customer WHERE CustNum = 1000. /* Raises ERROR and execution goes to
                                           CATCH block. */

    MESSAGE "This message never appears because of ERROR condition."
        VIEW-AS ALERT-BOX BUTTONS OK.

    CATCH eSysError AS Progress.Lang.SysError:
        < Handler code for SysError condition >
        MESSAGE "Inside CATCH block." VIEW-AS ALERT-BOX BUTTONS OK.
        /* Execution goes to FINALLY before leaving DO block. */
    END CATCH.

    FINALLY:
        < Your code >
```

```
        MESSAGE "Inside FINALLY block." VIEW-AS ALERT-BOX BUTTONS OK.
        /* LEAVE DO block here. */
    END FINALLY.

END. /* DO */

MESSAGE "Out of DO block." VIEW-AS ALERT-BOX BUTTONS OK.
```

If you run this code, you see the following messages:

```
Inside CATCH block.
Inside FINALLY block.
Out of DO block.
```

Example 3

In Example 3, after ERROR is raised, execution goes to the CATCH block, which rethrows the error. The FINALLY block executes for the DO block before the ERROR is raised in the procedure block. The MESSAGE statement there does not execute because of the raised error, but the outer FINALLY runs.

```
DO ON ERROR UNDO, LEAVE:
    FIND Customer 1000. /* Raises ERROR and execution goes to the CATCH block. */

    MESSAGE "This message never appears because of ERROR condition."
        VIEW-AS ALERT-BOX BUTTONS OK.

    CATCH eSysError AS Progress.Lang.SysError:
        < Handler code for SysError condition >
        MESSAGE "Inside CATCH block."
            VIEW-AS ALERT-BOX BUTTONS OK.
        /* Execution goes to FINALLY before leaving DO block. */
        UNDO, THROW eSysError.
    END CATCH.

    FINALLY:
        < Your code >
        MESSAGE "Inside inner FINALLY block."
            VIEW-AS ALERT-BOX BUTTONS OK.
    END FINALLY.

END. /* DO */

MESSAGE "This message never appears because of ERROR thrown from CATCH block."
<other code>

FINALLY:
    < Your code >
    MESSAGE "Inside outer FINALLY block."
        VIEW-AS ALERT-BOX BUTTONS OK.
END FINALLY.
```

If you run this code, you see the following messages:

```
Inside CATCH block.
Inside inner FINALLY block.
Inside outer FINALLY block.
```

FINALLY blocks and STOP-AFTER

If STOP is raised because of a STOP-AFTER phrase, a FINALLY block still runs, just like for any other STOP-AFTER condition.

If a STOP-AFTER phrase is in effect, but has not timed out, the time it takes to run any FINALLY blocks is incorporated into the time elapsed, as with any other code. However, if the FINALLY block is executing when the time elapses, this does not raise a STOP condition. The FINALLY block, and any sub-blocks or routines that it executes, run to completion. When the FINALLY block completes, if the associated block is the block containing the STOP-AFTER phrase, whose time has elapsed, the STOP condition is not raised for that block, since the block is already over. However, if the STOP-AFTER phrase is on an outer block, it is still in effect when the FINALLY block completes. So now that the time has elapsed, the STOP condition is raised

If a STOP-AFTER phrase is used in the FINALLY block itself or in any sub-block, procedure, method, user-defined function, or property accessor called from the FINALLY block, it is ignored.

Conflicts between the associated and FINALLY blocks

It is possible for a statement that ends a FINALLY block to conflict with a statement that ends the associated block. In these scenarios the general rule is that the last action wins. The following examples illustrate some of these cases:

Example scenario	Result
The associated block of a function or non-void method returns a value (for example, `RETURN 5.`) and then the FINALLY block executes a conflicting RETURN statement (for example, `RETURN 10.`).	Returns 10 from the function or non-void method.
The associated block raises an error to the outer block or caller and then the FINALLY block returns a value (for example, `RETURN 10.`).	Returns 10 from the function or non-void method rather than raise the error.
The associated block returns a value (for example, `RETURN 5.`) and then the FINALLY block throws an error to the outer block or caller.	Raises an error in the outer block or caller; thus the original return value of 5 is lost.

Best programming practices avoid these conflict scenarios. For example, there should only be one RETURN statement to return a value for any code path. If there is a possibility that the FINALLY block can raise ERROR, usually this is not as important as the original error from the associated block. Therefore, it is a good practice to use a CATCH block or NO-ERROR in the FINALLY block to handle it, so the original error propagates up.