OpenEdge® Development: Error Handling
Notices

© 2014 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.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Easy!, Fathom, Making Software Work Together, OpenEdge, Powered by Progress, Progress, Progress Control Tower, Progress RPM, Progress Software Business Making Progress, Progress Software Developers Network, Rollbase, RulesCloud, RulesWorld, SequeLink, SpeedScript, Stylus 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. AccelEvent, AppsAlive, AppServer, BusinessEdge, DataDirect Spy, DataDirect SupportLink, , Future Proof, High Performance Integration, Modulus, OpenAccess, Pacific, ProDataSet, Progress Arcade, Progress Pacific, Progress Profiles, Progress Results, Progress RFID, Progress Progress Software, ProVision, PSE Pro, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress 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.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

The Release Notes can be found in the OpenEdge installation directory and online at: https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx.

For the latest documentation updates see OpenEdge Product Documentation on Progress Communities: (https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx).

December 2014

Last updated with new content: Release 11.4.0

OpenEdge Development: Error Handling
# Table of Contents

**Preface**
- Purpose ................................................................. 9
- Audience ............................................................. 9
- Organization ......................................................... 10
- Using this manual .............................................. 10
  - References to ABL compiler and run-time features .... 11
  - References to ABL data types .......................... 11
- Typographical conventions .................................. 11
- Examples of syntax descriptions .......................... 13
  - Long syntax descriptions split across lines ........... 14
  - Complex syntax descriptions with both required and optional elements .... 15
- Example procedures ............................................. 15
- OpenEdge messages ............................................. 16
  - Obtaining more information about OpenEdge messages .... 17

**Chapter 1: Introduction to Condition Handling** .................. 19
- How to use this manual ........................................ 20
  - New ABL users ................................................. 20
  - ABL users familiar with traditional error handling .... 21
  - ABL users interested in structured error handling .... 21
- Conditions ........................................................... 21
  - ERROR .......................................................... 22
  - ENDKEY ......................................................... 22
  - STOP ............................................................ 22
  - QUIT ........................................................... 22
- Traditional error handling ..................................... 23
  - Default error handling ..................................... 23
  - Changing block error handling .......................... 24
  - Suppressing the ERROR condition .................... 24
  - Custom error handling ..................................... 24
  - Application error handling ............................... 25
  - Conclusion ...................................................... 25
- Structured error handling ..................................... 25
  - What is structured error handling ....................... 26
  - What is ABL structured error handling ............... 27
  - Conclusion ...................................................... 27

**Chapter 2: ABL Block Essentials** .................................... 29
Chapter 3: Traditional Error Handling

Statement errors
Statement behavior versus block behavior
Understanding OpenEdge messages
Progress messages (promsgs) file
Specifying error handling behavior
Using labels
Suppressing errors
Providing custom error handling
Testing for statement errors
Testing for built-in method warnings
Undoing work
Raising the ERROR condition
User defined functions

Chapter 4: STOP and QUIT Condition Handling

STOP condition handling
Default handling
Example: default handling for lost database connections
Raising the STOP condition
Overriding default handling
Raising a timed STOP condition
QUIT condition handling
Default handling
Raising the QUIT condition
Overriding default handling

Chapter 5: Using Structured Error Handling

Comparison to traditional error handling
Additional language features
Structured error handling tasks
Next steps
## Chapter 6: Using ABL Error Classes

- **Progress.Lang.Error** interface........................................................................................................... 70
  - Throwing class-based error objects from an AppServer to an ABL client..................................................... 71
- **Progress.Lang.ProError** class................................................................................................................. 72
- **Progress.Lang.SysError** class.................................................................................................................. 72
- **Progress.Lang.SoapFaultError** class........................................................................................................... 73
- **Progress.Lang.AppError** class................................................................................................................... 74
  - Properties and methods................................................................................................................................. 75
  - Constructors.............................................................................................................................................. 76
  - Example..................................................................................................................................................... 77
  - Enabling stack tracing with error objects................................................................................................... 79

## Chapter 7: Handling Errors with CATCH Blocks

- **Introduction**............................................................................................................................................ 82
- **CATCH** statement syntax.......................................................................................................................... 83
- **Blocks that support CATCH blocks**.......................................................................................................... 86
- **UNDO** scope and relationship to a CATCH block....................................................................................... 87
- **Buffer scope within a CATCH block**.......................................................................................................... 89
- **Flow-of-control statements in a CATCH block**.......................................................................................... 91
- **CATCH** blocks within **CATCH** blocks.................................................................................................. 93
- **System error message suppression with CATCH**....................................................................................... 94
- **Handling errors from built-in ABL methods**.............................................................................................. 95

## Chapter 8: Raising errors with THROW

- **Propagating errors up the stack**................................................................................................................ 97
- **THROW** with error objects........................................................................................................................ 98
  - **THROW** compared with **RETURN** ERROR......................................................................................... 99
  - **THROW** as a default for blocks.................................................................................................................. 99
  - **BLOCK-LEVEL ON ERROR** UNDO, **THROW** statement................................................................... 99
    - Example.................................................................................................................................................. 100
  - **ROUTINE-LEVEL ON ERROR** UNDO, **THROW** statement............................................................ 101
    - -undothrow n startup parameter.............................................................................................................. 102
  - Determining error-handling characteristics of r-code................................................................................ 103
  - **UNDO-THROW-SCOPE** attribute........................................................................................................ 103
  - **XREF** and **XREF-XML** output.......................................................................................................... 103
  - **THROW** with user-defined functions.................................................................................................... 104
  - Throwing system errors............................................................................................................................... 104
  - Throwing application errors....................................................................................................................... 106

## Chapter 9: Using FINALLY End Blocks

**Introduction.............................................................................................................................................. 107**
Syntax.................................................................................................................................................108
Behavior..............................................................................................................................................109
  Execution of a FINALLY block................................................................................................109
  Transaction scope in a FINALLY block..................................................................................109
  Buffer scope in a FINALLY block.........................................................................................109
  Flow-of-control in a FINALLY block....................................................................................110
  STOP or QUIT conditions with FINALLY blocks.................................................................110
Examples.............................................................................................................................................110
  Example 1.............................................................................................................................110
  Example 2.............................................................................................................................111
  Example 3.............................................................................................................................112

Index..................................................................................................................................................115
Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Using this manual
- Typographical conventions
- Examples of syntax descriptions
- Example procedures
- OpenEdge messages

Purpose

This book provides a programming guide for ABL error handling.

Audience

This book is intended for all ABL programmers.
Organization

Introduction to Condition Handling on page 19
Provides an overview of the two ABL error handling models: traditional error handling and structured error handling.

ABL Block Essentials on page 29
Summarizes essential block information critical to understanding error handling use cases.

Traditional Error Handling on page 37
Provides complete documentation on the traditional error handling model.

STOP and QUIT Condition Handling on page 51
Documents the STOP and QUIT conditions.

Using Structured Error Handling on page 63
Describes how structured error handling is both a new model and a model that expands the features of traditional error handling.

Using ABL Error Classes on page 69
Describes the hierarchy of built-in ABL classes that represent error objects.

Handling Errors with CATCH Blocks on page 81
Provides in-depth information on using CATCH block to handle errors.

Raising errors with THROW on page 97
Provides more complex use cases for the THROW directive.

Using FINALLY End Blocks on page 107
Describes how to use the FINALLY block for end of block processing.

Using this manual

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 see the OpenEdge Product Documentation Overview page on Progress Communities:

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 manual uses the following typographical and syntax conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td>Convention</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SMALL, BOLD CAPITAL LETTERS</td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td>KEY1+KEY2</td>
<td>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.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 . . . ] }
```

Example procedures

This manual may provide example code that illustrates syntax and concepts. You can access many of the example files, and details for installing them, from the following locations:

- A self-extracting Documentation and Samples file available on the OpenEdge download page of the Progress Software Download Center
- The OpenEdge Product Documentation Overview page on Progress Communities:


Once installed, you can locate the example files for this manual in the following path under the OpenEdge Documentation and Samples installation directory:

<table>
<thead>
<tr>
<th>This directory . . .</th>
<th>Contains examples for the following documents . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>src\prodoc\dotnetobjects</td>
<td>OpenEdge Development: GUI for .NET Programming</td>
</tr>
<tr>
<td>src\prodoc\dynamics</td>
<td>The Progress Dynamics documentation</td>
</tr>
<tr>
<td>src\prodoc\getstartoop</td>
<td>OpenEdge Development: Object-oriented Programming</td>
</tr>
<tr>
<td>src\prodoc\handbook</td>
<td>OpenEdge Getting Started: ABL Essentials</td>
</tr>
<tr>
<td>src\prodoc\interfaces</td>
<td>OpenEdge Development: Programming Interfaces</td>
</tr>
<tr>
<td>src\prodoc\json</td>
<td>OpenEdge Development: Working with JSON</td>
</tr>
</tbody>
</table>
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.
Introduction to Condition Handling

Condition handling, error handling, and exception handling are all industry terms for programming designed to respond to run-time occurrences that interrupt the usual flow of a software application. Depending on the language, these occurrences can be called conditions, errors, or exceptions. In ABL, condition refers to all possible flow interruptions. Error refers to the most common type of condition. An error occurs when the ABL virtual machine (AVM) fails to successfully execute ABL code. Errors raised by the AVM are called system errors. Error raised programatically are called application errors. In some languages, unrecoverable errors are called exceptions. In ABL, the STOP condition represents unrecoverable errors.

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

To be precise, the topic of this manual is condition handling. Since most condition handling in ABL involves the ERROR condition, error handling is a defacto synonym for condition handling and the most frequently used term in the OpenEdge documentation set.

For details, see the following topics:

- How to use this manual
- Conditions
- Traditional error handling
- Structured error handling
How to use this manual

To get the most out of this manual, you need to know that ABL has two error handling models and this manual is organized around describing these models separately. Your interest in either or both of these models is your best guide to understanding which parts of the manual will be useful to you. The minimal definitions of the models below will guide you as you get started:

- **Traditional error handling** — All error handling features included in ABL before the release of OpenEdge Release 10.1C comprise the ABL’s original error handling model. To allow useful comparisons between original and newer error handling, the documentation refers to the original error handling features and techniques as *traditional error handling*.

  Traditional error handling is still a viable error handling model and comprises the foundation of the new error handling model. In fact, original and new error handling features seamlessly coexist and interoperate in ABL code.

- **Structured error handling** — Introduced in OpenEdge Release 10.1C, *structured error handling* builds on existing error handling features. It represents errors as class-based objects that can be trapped with custom error handlers and easily passed from a called context to a calling context. Structured error handling is sometimes referred to as *try and catch* functionality and is a common feature in object-oriented programming languages.

The content of this chapter is essential for all users. The rest of the chapters are likely to interest you depending on which of the following groups you belong to:

- **New ABL users** on page 20
- **ABL users familiar with traditional error handling** on page 21
- **ABL users interested in structured error handling** on page 21

**New ABL users**

New ABL users will find essential material in every chapter. You should read the introductory chapters on structured error handling even if traditional error handling seems like what you need right now. While structured error handling is an object-oriented feature set, it provides powerful, easy-to-learn, and easy-to-program functionality for any ABL application. You do not have to commit to larger object-oriented design patterns to take advantage of structured error handling.

To better 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.
ABL users familiar with traditional error handling

You can find the programming documentation on traditional error handling beginning in this chapter and concluding with STOP and QUIT Condition Handling on page 51.

To determine if you can benefit from structured error handling, see Using Structured Error Handling on page 63. Note that ABL options specifically for structured error handling are always indicated as such. Syntax boxes show always the complete syntax, and point readers to the location of the information on the structured error handling options if documented elsewhere.

Discussions of how traditional and structured error handling code interoperate are found throughout in the manual. To quickly find details about a particular language element, see OpenEdge Development: ABL Reference. You can also check the index of this manual for relevant notes on ABL language elements.

ABL users interested in structured error handling

You can skip several chapters if you are comfortable with the following topics:

- Block properties including default error handling
- Using the ON ERROR phrase
- Using the RETURN ERROR statement
- Using the NO-ERROR option
- Using the ERROR-STATUS system handle
- ABL error messages

After reading this chapter and skimming ABL Block Essentials on page 29 you can then skip to Using Structured Error Handling on page 63.

If your interest in structured error handling includes developing a hierarchy of classes to represent your custom application error types, then first read OpenEdge Development: Object-oriented Programming.

Conditions

ABL recognizes four conditions, all of which are keywords:

- ERROR
- ENDF KEY
- STOP
- QUIT
ERROR

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

- The AVM fails to execute ABL code, usually at the statement level. 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. An error raised in this way is called an application error.
- An application user presses the key mapped to the **ERROR** keycode. This style of user interface error has been replaced in modern programming by UI triggers.

ENDKEY

The **ENDKEY** condition represents run-time occurrences that have significant meaning for an older style of programming that features tightly coupled UI and database record manipulation. Because applications should not normally mix data input with transaction blocks in the same procedures, the **ENDKEY** condition is not frequently used in modern applications. The **ENDKEY** condition occurs when:

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

Related to the **ENDKEY** condition is another keycode called **END-ERROR**. **END-ERROR** detects more special cases related to the older style of programming and provides default processing.

If you have interest in **ENDKEY** and **END-ERROR** processing, refer to the **ON ENDKEY** and **ON END-ERROR** phrase entries in *OpenEdge Development: ABL Reference*.

STOP

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

- Your application executes a **STOP** statement.
- An application user presses the key mapped to the **STOP** keycode when input is enabled.
- The AVM encounters a system error that is deemed unrecoverable. For example, if the AVM detects a lost database connection or cannot find a procedure file specified in a **RUN** statement, the AVM raises the **STOP** condition.

QUIT

The **QUIT** condition only occurs when your application executes a **QUIT** statement.
Traditional error handling

This section provides a brief overview of the concepts and mechanics of traditional error handling. Traditional Error Handling on page 37 provides complete documentation on the topics introduced here.

Default error handling

Default error handling is defined at the block level and is considered a property of the block. It can vary depending on the block type and whether the work inside the block is part of a transaction. Consider this single line procedure:

```
FIND FIRST Customer WHERE CustNum = 1000.
```

This statement will fail since the Sports2000 database does not contain a matching Customer record. When it fails, the AVM performs the following steps:

1. The AVM stops execution of the block at the failed statement.

2. The AVM displays a message (like the one shown) to the default output destination, which is usually the screen:

   ![Error Message]

3. The AVM checks the block that encloses the failed statement to get the default error handling instructions.

4. Error handling always begins with the UNDO action. The AVM undoes any changes to undoable variables and temp-table fields. Variables and temp-table fields are undoable by default unless they were defined with the NO-UNDO option. The AVM also undoes the current transaction, if one exists. A transaction exists if the block:

   - Can change database fields
   - Has the TRANSACTION keyword in the block header statement
   - Is a sub-block of a block that is itself a transaction.

See the chapter on transactions in OpenEdge Getting Started: ABL Essentials to review transaction fundamentals.

In this example, there is nothing that needs to be undone.
5. The AVM attempts the default branching option. This could be RETRY, LEAVE, or NEXT, depending on the context. (Branching options are described fully in Traditional Error Handling on page 37. In this example, the branching option is LEAVE.

6. The AVM terminates the procedure and returns to the operating system or the OpenEdge tool from which the procedure was run.

Changing block error handling

The ON ERROR phrase lets you change the error handling for a DO, FOR, or REPEAT block. For example:

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

If this code were included in an internal procedure or other callable routine, then the failure of the FIND statement leads the AVM to the ON ERROR phrase and the RETURN option. RETURN forces the called routine to terminate and return to the caller.

Suppressing the ERROR condition

Many ABL statements support the NO-ERROR option. The NO-ERROR option directs the AVM to redirect any failure that would have otherwise raised ERROR. Executing the following example does nothing, and the procedure terminates normally:

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

Information about errors that occurred on a statement with the NO-ERROR option are redirected to the ERROR-STATUS system handle. The MESSAGE statement retrieves the first redirected error message from the system handle and displays it in the message area of the procedure window.

Custom error handling

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

- Override the default error handling
- Suppress your customer from seeing the default error message
- Provide custom code to handle the particular system error
- Test the ERROR-STATUS system handle to determine if a particular error occurred
In this simple example, the custom error handling creates a custom error message by outputting the standard error message to the run-time message area of the procedure window:

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

**Application error handling**

You can raise the `ERROR` condition yourself with the `RETURN ERROR` statement and force the default error handling of the block to execute. The default error handling occurs after execution resumes in the caller.

You can also specify a character string with `RETURN ERROR` that represents your application error data. You access the returned string with the `RETURN-VALUE` function.

In the following example, the code suppresses the default error handling in both the internal procedure and in the main procedure. It essentially replaces the standard error message and returns the custom message to the caller to be displayed after the `RUN` statement, as shown:

```
PROCEDURE Customer1000:
    FIND FIRST Customer WHERE CustNum = 1000 NO-ERROR.
    IF ERROR-STATUS:ERROR = TRUE THEN
        RETURN ERROR "Customer 1000 does not exist in this database."
    END IF.
END PROCEDURE.
RUN Customer1000 NO-ERROR.
IF ERROR-STATUS:ERROR = TRUE THEN
    DISPLAY RETURN-VALUE FORMAT "x(60)"
ELSE DISPLAY "Procedure completed successfully." FORMAT "x(60)"
```

**Conclusion**

The simple code examples shown provide a sketch of traditional error handling mechanics. See Traditional Error Handling on page 37 for complete documentation about each of the language elements shown and the default and programmed error handling behavior available in each use case.

**Structured error handling**

Structured error handling is an expansion of the original ABL error handling features, which are now referred to as *traditional error handling* features. While the new language features do provide you with a more robust error handling model, they coexist and interoperate with the traditional error handling features. Existing code will continue to work as is, and code using the new model can work with this existing code.
While both the new and the old language elements are now part of your set of error handling tools, the structured methodology that goes with the new model provides benefits that you will want to exploit immediately in your new programming projects.

What is structured error handling?

Structured error handling is an error handling model found in many languages, but is typically associated with object-oriented languages. It is characterized by the following features:

- Represents all errors as objects (instances of a class)
- Allows you to define your own error types (classes)
- Allows you to explicitly raise (throw) an error
- Allows you to handle (catch) particular error types in a particular context (ABL block)
- Allows you to propagate (re-throw) errors from the current context (inner block) to the immediate outer context (outer block). In other words, when an error occurs in a called context, you can direct the calling context to handle the error
- Allows you to specify code that executes at the conclusion of some associated code, whether or not the associated code executed successfully (FINALLY block)

Structured error handling has several syntactical variations, but you may know it by one common syntax: try and catch. (Italics indicate conceptual language elements as opposed to actual ABL elements.) Examine this pseudo code:

```
TRY /* to execute this code */
{
    WRITE FILE myAddresses.
}
CATCH errorFileLocked /* and if the errorFileLocked error is thrown (raised), let the error handling behavior here catch (handle) the error instead of the system's default behavior. */
{
    SEND MESSAGE "File currently unavailable."
}
CATCH errorNoWritePermission /* or if the errorNoWritePermission error is thrown, use this behavior.*/
{
    SEND MESSAGE "You do not have permission to write files."
}
```

The try statement provides the ability to define a block of code to which you can associate error handling behavior. The catch statement allows you to define an error handler and associate it with an error type. When the system detects an error in a try block, it will execute the code in the catch block that matches the error. What does it do if there is no matching catch block? The behavior in this scenario is language dependent, but most often, the call stack is unwound until the system finds a suitable catch block.

Thus, structured error handling turns an error condition into a named object that can be more than just an identifier and message. It provides structure by encouraging code to be defined in blocks associated with alternate, error-handling code paths (catch blocks). The syntactical structure provides an easy to read map of your error handling behavior.
This description of structured error handling describes the basic mechanism. Other features and default behaviors vary by language. The next step is to introduce the basic mechanism of ABL structured error handling before beginning a more thorough description in the rest of this manual.

What is ABL structured error handling?

Since ABL is a block structured language, ABL has no need for a block-making construct like the try statement. ABL blocks have something more than simple try blocks. Because ABL is a language with database semantics, all ABL blocks are undoable blocks that protect the database by undoing work when a block fails. This ability means you have more options for successfully resuming execution after an error. For example, you could retry a block after an error.

All blocks have implicit error handling of some type, except for the simple DO block (a DO ... END block without transaction or error handling options). Explicit error handling for blocks is provided by an ON ERROR phrase and its many options. Default error handling is provided by an implicit ON ERROR phrase. Implicit and explicit ON ERROR phrases make up, in brief, traditional error handling.

ABL adds built-in classes to represent errors as objects and a CATCH statement. These changes provide the basic mechanics of structured error handling in ABL. Here is a simple example:

```abl
DO ON ERROR UNDO, RETURN:
    FIND FIRST Customer WHERE CustNum = 1000.
    CATCH eSystemError AS Progress.Lang.SysError:
        MESSAGE "Not a valid customer number."
        UNDO, THROW eSystemError.
    END CATCH.
END. /* DO */
```

The explicit ON ERROR phrase defines error handling for the block. In traditional error handling, the RETURN action occurs if any error is raised within the block. In structured error handling, the RETURN action occurs for any error that is not explicitly handled by a CATCH block. Since the FIND statement fails and raises a system error, the CATCH block executes and the MESSAGE statement displays a customer error message in the message area of the procedure window. However, the CATCH block concludes with the UNDO, THROW statement that directs the block enclosing the DO block to handle the eSystemError object. The enclosing block is the main procedure block, which by default uses traditional error handling. So, the default error message will also appear in a message block when the error object is thrown to the main procedure block. This demonstrates how error handling moves seamlessly between traditional and structured error handling.

Conclusion

This section was just a quick introduction to the concepts of structured error handling and how those concepts are implemented in ABL. A complete tour of the model begins with Using Structured Error Handling on page 63.
ABL Block Essentials

ABL error handling starts at the statement level and is then handled by the nearest enclosing block. Default block properties and block statement options then determine how the block handles the error. This chapter summarizes essential block information in a way that provides a useful framework for remembering the many error handling use cases discussed in this manual.

For details, see the following topics:

- Block types
- Block summary
- Understanding the UNDO concept
- Branching options
- The UNDO statement

Block types

An ABL block is a set of ABL statements executed as a unit. A block is defined with a block statement, a body (the set of ABL statements), and an END statement. The execution of the block body is determined by the function of the block type, the default properties of the block, and the block statement options in effect. All ABL statements must be contained by a block.

ABL blocks can be grouped into these types:

- Basic blocks on page 30
- End blocks on page 30
Basic blocks

Basic blocks provide fundamental procedural functionality. Although they provide flow-of-control (branching) options, they are not strictly callable units and are, therefore, not routines. The basic blocks are:

- DO
- FOR
- REPEAT

These three blocks have a rich set of options that allow a larger set of basic block variations. Particular variations are often treated as individual blocks in ABL documentation where a variation is important to the current topic. For example, the FOR EACH block is important to batch record processing and the DO TRANSACTION block is important for defining a transaction or subtransaction where ABL would not automatically do so.

Here are some important points about basic blocks:

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

End blocks

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

- CATCH
- FINALLY

Here are some important points about end blocks:

- The CATCH block defines a custom error handling block and is strictly a feature of the structured error handling model.
- The FINALLY block contains code that must be executed at the conclusion of the block (or block iteration) whether the associated block completed successfully or raised error. The FINALLY block is an important feature for structured error handling, but it can be just as useful with traditional error handling. Using FINALLY blocks with traditional error handling does not alter the behavior of the traditional error handling model.
If used, the **FINALLY** block must be the last end block.

**Routine-level blocks**

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

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

The routine-level blocks are:

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

Here are some important facts about routine-level blocks:

- A procedure file has an implicit block that contains all the code contained in the procedure file. Think of the implicit block as an invisible block statement (header) and an invisible **END** statement. End blocks placed at the end of a procedure file belong to this implicit procedure block.
- While internal procedures and user-defined functions are defined in a procedure block, at runtime these routine blocks become sub-blocks of the code that invokes the routine. In other words, the routines are not scoped to the procedure block.
- A database trigger procedure file (.p) is a block with an explicit block header statement and an implicit **END** statement. Place end blocks at the very end of the trigger procedure file. The compiler issues an error if you add an explicit **END** statement after the end blocks.
- A UI trigger can be "called" by using the **APPLY**event-name syntax.

**Class block**

The class block (.cls), which is the root of ABL object-oriented programming, is not executable in the same sense as the other ABL blocks, which are procedural in nature. The class block defines an abstract object and its members. Within the class block, the following types of class members are executable routine-level blocks:

- User-defined methods
- Constructors
- Destructors
- Property accessors (GET and SET methods of a property)

## Block summary

For each ABL block (and pertinent variety), the following table lists its common name, type, and the ABL statement used to define it.

### Table 1: ABL block types

<table>
<thead>
<tr>
<th>Block name</th>
<th>Type</th>
<th>Defined by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple DO block</td>
<td>Basic</td>
<td>DO statement without either the TRANSACTION option or the ON ERROR phrase</td>
</tr>
<tr>
<td>DO TRANSACTION block</td>
<td>Basic</td>
<td>DO statement with the TRANSACTION option</td>
</tr>
<tr>
<td>DO ON ERROR block</td>
<td>Basic</td>
<td>DO statement with the ON ERROR phrase</td>
</tr>
<tr>
<td>FOR block</td>
<td>Basic</td>
<td>FOR statement</td>
</tr>
<tr>
<td>REPEAT block</td>
<td>Basic</td>
<td>REPEAT statement</td>
</tr>
<tr>
<td>CATCH block (Structured error handling only)</td>
<td>End</td>
<td>CATCH statement</td>
</tr>
<tr>
<td>FINALLY block (Traditional and structured error handling)</td>
<td>End</td>
<td>FINALLY statement</td>
</tr>
<tr>
<td>Procedure (External procedure)</td>
<td>Routine-level(File)</td>
<td>The implicit all-enclosing block of a procedure file</td>
</tr>
<tr>
<td>Internal procedure</td>
<td>Routine-level</td>
<td>PROCEDURE statement</td>
</tr>
<tr>
<td>User-defined function</td>
<td>Routine-level</td>
<td>FUNCTION statement</td>
</tr>
<tr>
<td>Database trigger procedure</td>
<td>Routine-level(File)</td>
<td>TRIGGER PROCEDURE statement in a procedure file</td>
</tr>
<tr>
<td>Database trigger block</td>
<td>Routine-level</td>
<td>ON statement with a database event specified</td>
</tr>
<tr>
<td>User-interface trigger</td>
<td>Routine-level</td>
<td>ON statement with a user-interface event specified</td>
</tr>
<tr>
<td>Class method (User-defined method)</td>
<td>Routine-level</td>
<td>METHOD statement</td>
</tr>
<tr>
<td>Class constructor</td>
<td>Routine-level</td>
<td>CONSTRUCTOR statement</td>
</tr>
<tr>
<td>Class destructor</td>
<td>Routine-level</td>
<td>DESTRUCTOR statement</td>
</tr>
</tbody>
</table>
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. Undoing is essentially throwing away the current transaction.

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 undoable variable data occurs in block, the AVM undoes changes to these variables and fields, whether or not the block is a transaction block. Since providing undo behavior for variable and temp-table data incurs additional overhead, define variables and temp-tables fields with the NO-UNDO option when possible. With NO-UNDO in effect, the AVM will not allocate the resources needed to track changes, and any undo action ignores the NO-UNDO data items.

Transactions are scoped to blocks. Default error handling is also scoped to blocks, and the first step in default error handling is to undo. Thus, undoing a transaction might seem to be synonymous with undoing a block. Actually, the only work being undone is changes to database fields, undoable variables, and temp-table fields. Other actions caused by statements and routines are not effected.

A block is a transaction block if the block 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.
Note: This introduction to undo touched on related transaction concepts. Understanding transactions is an important prerequisite to understanding default error handling. Transaction information in this chapter describes some default transaction behavior and always 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 example, is an iteration of a nested block a complete transaction, or is it merely a subtransaction of the transaction defined by the enclosing block? The answer to questions like these will help you understand which error handling options will be most useful for your applications. For more information, see the chapter on transactions in *OpenEdge Getting Started: ABL Essentials*.

**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 options (flow of control options) as defined below:

- **RETRY** — If a block contains user input logic, 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. Since good modern programming practice recommends separating user interface logic from transaction logic, RETRY is not normally used.

- **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 to do, 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 quits. RETURN has many options itself and is fully discussed in Traditional Error Handling on page 37.

- **THROW** — (Structured error handling) Indicates that the AVM should suppress the default error message, exit the block, and pass an error object to the caller or the next enclosing block.

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

**Table 2: Default branching on error**

<table>
<thead>
<tr>
<th>Block Type</th>
<th>If user input detected</th>
<th>If iterating</th>
<th>Otherwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO TRANSACTION</td>
<td>RETRY</td>
<td>NEXT</td>
<td>LEAVE</td>
</tr>
<tr>
<td>FOR</td>
<td>RETRY</td>
<td>NEXT</td>
<td>–</td>
</tr>
<tr>
<td>REPEAT</td>
<td>RETRY</td>
<td>NEXT</td>
<td>LEAVE</td>
</tr>
<tr>
<td>End blocks</td>
<td>–</td>
<td>–</td>
<td>THROW</td>
</tr>
<tr>
<td>Routine-level blocks</td>
<td>RETRY</td>
<td>–</td>
<td>LEAVE</td>
</tr>
<tr>
<td>Trigger procedure file</td>
<td>RETRY</td>
<td>–</td>
<td>RETURN ERROR</td>
</tr>
</tbody>
</table>
The UNDO statement

You might also want to undo a transaction if your business logic detects errors other than database errors. The UNDO statement allows you to undo the current transaction and specify a branching action. This is the syntax for the UNDO statement:

Syntax

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

Notice the label syntax. If your block header statements define labels for your blocks, then you can use those labels to control which block gets undone or which block to perform a LEAVE or NEXT branch to. Labels let you control how much of the transaction to undo when you are working with nested blocks.

**Note:** If you use the LEAVE label2 or NEXT label2 option, label2 cannot be a block outside the scope of the UNDO.

The remaining UNDO options will be covered in Traditional Error Handling on page 37.
ABL has always been a procedural, block-structured, database language with the ability to mix procedural, database, and user-interface logic in a single procedure file of statements, blocks, and nested blocks. The semantics of working with a database means ABL error handling protects persistent data above all with tight integration to transaction management. Error handling draws on ABL procedural strengths to provide flow-of-control options that increase the chance of restoring application flow after an error.

ABL has expanded to provide object-oriented programming, distributed programming, and extensive integration features with other features of the OpenEdge platform. Consequently, there are many specialized error-handling topics. The traditional error handling model provides an easy-to-use set of features for general ABL programming. It also provides a standard way of communicating errors to ABL from integrated OpenEdge features. For example, a Web service returns errors as industry standard SOAP faults. The traditional error handling model lets your application handle a returned SOAP fault as an ABL system error.

To have a good working knowledge of the traditional error handling model, you should:

- Know how a particular statement handles an error
- Understand OpenEdge messages
- Know how a particular block handles an error (see ABL Block Essentials on page 29)
- Know how to use block statement options to alter default error handling
- Know how to suppress system errors with the NO-ERROR option
- Know how to test for errors after a NO-ERROR option and branch to your custom error handling logic
- Know how to raise ERROR and specify an application (custom) error data
- Know how to explicitly undo transactions
• Understand STOP and QUIT condition processing (see STOP and QUIT Condition Handling on page 51)

This chapter provides task-oriented documentation on the ABL elements that comprise the traditional error handling model, organized and described in this chapter. For details, see the following topics:

• Statement errors
• Understanding OpenEdge messages
• Specifying error handling behavior
• Suppressing errors
• Providing custom error handling
• Undoing work
• Raising the ERROR condition

Statement errors

In ABL, when the AVM is not able to properly execute a statement, the AVM most often raises the ERROR condition. From the examples you saw in Introduction to Condition Handling on page 19 a FIND statement that fails to find a matching record raises the ERROR condition. If the statement failure is serious enough, the AVM may raise the STOP condition instead. For example, a RUN statement that cannot find a specified procedure file raises the STOP condition. In some cases, the statement will not raise the ERROR condition. For example, the MESSAGE statement does not raise ERROR.

In these examples, the different responses to unexpected run-time occurrences do not follow a pattern that can be summarized by a single rule. The design of each ABL statement includes decisions about how to respond to run-time conditions. The design focus is always on the appropriateness of raising a condition, given the purpose of the statement. For example:

• A failed FIND statement means that record buffer data is not available for the data accessing statements that are sure to follow a FIND. So, raising the ERROR condition is appropriate.

• A failed RUN statement indicates missing application functionality, so stopping the application is appropriate.

• The MESSAGE statement provides troubleshooting information or user feedback. Therefore, it is appropriate to display as much of the message as possible, even if the AVM could not resolve one or more output expressions. So, it is appropriate that the MESSAGE statement does not raise error.

OpenEdge Development: ABL Reference documents how each statement responds to run-time conditions. When using an ABL statement for the first time, a quick check of the reference entry will clarify how to handle possible errors. This is especially true for more specialized statements and system object methods. Some of these language elements may have additional error handling behavior or functionality beyond the standard functionality of the traditional error handling model.
Statement behavior versus block behavior

Consider this procedure:

```c
/* This statement executes and raises the ERROR condition. */
FIND FIRST Customer WHERE CustNum = 1000.

/* This statement does not execute. */
FIND FIRST Customer WHERE CustNum = 2000.
```

The first statement will fail since the Sports2000 database does not contain a matching `Customer` record. When it fails, the AVM stops execution of the block at the failed statement. In other words, if another executable statement follows a failed statement in a block, it will not execute.

The AVM displays a message, like the following, to the default output destination, which is usually the screen:

![Error message example]

This also represents the line between error processing at the statement level and error processing at the block level. The AVM takes its next actions based on either default error handling block properties or statement options of the block header statement. Block statement options cannot alter the halt in execution or the display of the default error message. Only the `NO-ERROR` option on the statement can do this.

Recall from ABL Block Essentials on page 29 that even a simple procedure has an implicit block with an implicit `ON ERROR` phrase. The default error handling for that block is `UNDO` followed by either `RETRY` or `LEAVE`. In the previous example, there is no user input in the procedure, so the branching option is `LEAVE`. Control returns to the caller of the procedure block. Since there is no caller, the "application" terminates.

Understanding OpenEdge messages

Run-time system error messages inform you of errors encountered while the AVM is running a procedure.

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.

The Preface of each OpenEdge document, including this one, contains standard instructions for accessing information about specific error messages. You can use the Help system in the Progress Developer Studio for OpenEdge to get more information about particular error messages.
An OpenEdge error is associated with an error string and an error number, as shown in the following example:

<table>
<thead>
<tr>
<th>Error string</th>
<th>Error number</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIND FIRST/LAST failed for table Customer</strong> (565)</td>
<td></td>
</tr>
</tbody>
</table>

In OpenEdge documentation, the term *error message* refers to the error string and the error number together as displayed in the standard output.

**Progress messages (promsgs) file**

The OpenEdge platform stores its messages in a promsgs (Progress messages) file. Set up options for this file are described in *OpenEdge Getting Started: Installation and Configuration*.

You can find information on translation and localization of promsgs files in *OpenEdge Development: Internationalizing Applications*.

**Specifying error handling behavior**

The *ON ERROR* phrase is the ABL tool for altering the default error handling for basic blocks. This is the syntax for the *ON ERROR* phrase:

**Syntax**

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

]```

The syntax options are nearly identical to those of the *UNDO* statement shown in *ABL Block Essentials* on page 29.

*label1*

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

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 labeled with label1.

RETRY [label1]

Indicates that after undoing the processing of a block, the AVM repeats the same iteration of the block you name with the label1 option.

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

RETURN ...

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 labeled with label1.
Returns to the calling routine, or if there is no calling routine, returns to the OpenEdge Editor. The following table describes various RETURN cases:

**Table 3: RETURN cases**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return-value</td>
<td>The CHARACTER string you provide is passed to the caller. The caller can use the RETURN-VALUE function to read the returned value. For user-defined functions, the value must match the specified return type.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Raises ERROR in the caller and undoes the current subtransaction. You cannot specify ERROR within a user-interface trigger block or a destructor.</td>
</tr>
<tr>
<td>ERROR return-value</td>
<td>Raises ERROR in the caller and undoes the current subtransaction (except for user-defined functions). The CHARACTER string you provide is available to the caller in the RETURN-VALUE function. In structured error handling, the AVM also creates an AppError object and stores the return-value in the ReturnValue property.</td>
</tr>
<tr>
<td>ERROR error-object-expression</td>
<td>In structured error handling, raises ERROR in the caller and undoes the current subtransaction. The specified error object is your code. If it is an AppError object, the caller can also use the RETURN-VALUE function to read the setting of the ReturnValue property.</td>
</tr>
<tr>
<td>NO-APPLY</td>
<td>In a user-interface trigger, prevents the AVM from performing the default behavior for that event.</td>
</tr>
</tbody>
</table>

**THROW (structured error handling only)**

Use this directive to explicitly propagate an error to the enclosing block. This option is explained in detail in *Raising errors with THROW* on page 97.

The options of the ON ERROR phrase are parallel to the block properties that define default error handling for a block. All blocks, except the simple DO block, have an implicit ON ERROR phrase. So, for a procedure block, which does not support an explicit ON ERROR phrase, you can say that the procedure block has an implicit ON ERROR phrase. It is an implicit ON ERROR UNDO, LEAVE phrase if the procedure block does not contain user input. If the procedure block does contain user input, then it has an implicit ON ERROR UNDO, RETRY phrase.

Going forward, this manual uses the notion of an implicit (or explicit) ON ERROR phrase to describe the error handling properties of blocks. This terminology change makes it easy to compare different blocks and different use cases.
Using labels

The example that follows 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.

This version demonstrates how a label can be used to undo the transaction associated with the outer block rather than just the subtransaction of the inner block, as shown:

```plaintext
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:

- The ASSIGN statement in Outer-Block starts a transaction
- The FIND statement in Inner-Block raises the ERROR condition
- Inner-Block is automatically undone
- The explicit ON ERROR phrase of Inner-Block activates
- Outer-Block is undone
- Control returns to the caller, which is the main procedure block

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

<table>
<thead>
<tr>
<th>Block</th>
<th>ON ERROR phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure block (.p file)</td>
<td>Implicit ON ERROR UNDO, LEAVE</td>
</tr>
<tr>
<td>Internal procedure NestedBlocks</td>
<td>Implicit ON ERROR UNDO, LEAVE</td>
</tr>
</tbody>
</table>
When the AVM raises ERROR on the first iteration of Inner-Block, the explicit ON ERROR phrases directs the AVM to undo Outer-Block and use the RETURN branching option. RETURN anywhere in a routine forces control back to the caller, which is the procedure file. Since the RETURN option did not include the ERROR option, ERROR is not raised in the procedure block, and the final DISPLAY statement executes.

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

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

Raising ERROR in the procedure block returns control to the OpenEdge Editor.

If you change the explicit ON ERROR phrase to use the NEXT branching option, for each Customer record you will see one Order number displayed followed by an error message:

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

Finally, if you delete the explicit ON ERROR phrase, the implicit ON ERROR phrase for Inner-Block will be ON ERROR UNDO, NEXT. The UNDO operation will be applied to Inner-Block instead of Outer-Block. You will see one error message for each iteration of Inner-Block for each Customer record.

### Suppressing errors

The NO-ERROR option on an ABL statement suppresses all ABL error handling. For example:

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

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

Refer to OpenEdge Development: ABL Reference to see what particular statements support the NO-ERROR option. In some cases, the standard NO-ERROR option behavior may be augmented to suit a particular statement.
Providing custom error handling

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

The ERROR-STATUS system handle allows you to test whether any error occurred or whether a particular error occurred. You could have a branch that executes when a particular error occurs, and you could have another branch that executes when any other error occurs.

The attributes and methods of the handle allow you to make specific tests. If specific errors are important to you, the error numbers of the error message will be useful to you. If you need to repackage the standard error string to output it to a log file or other alternate destination, you can retrieve this information as well.

The following table describes the significant attributes and methods of the ERROR-STATUS system handle.
Table 5: ERROR-STATUS system handle attributes and methods

<table>
<thead>
<tr>
<th>Attribute or method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR attribute</td>
<td>If the ABL statement uses the <strong>NO-ERROR</strong> option and the AVM raises the <strong>ERROR</strong> condition, then this attribute is set to <strong>TRUE</strong>. If a method on an ABL system object generates an error message, the AVM does not raise error and this attribute remains <strong>FALSE</strong>. ABL treats built-in method errors as warnings rather than true errors.</td>
</tr>
<tr>
<td>ERROR-OBJECT-DETAIL attribute</td>
<td>If a Web service method returns a SOAP fault, the AVM stores the SOAP fault information in an ABL SOAP-fault object and raises <strong>ERROR</strong>. The AVM stores a handle reference to the SOAP-fault object in this attribute.</td>
</tr>
<tr>
<td>NUM-MESSAGES attribute</td>
<td>Provides an integer count of all the error messages generated by the statement with the <strong>NO-ERROR</strong> option.</td>
</tr>
<tr>
<td>GET-MESSAGE( index ) method</td>
<td>Allows you to retrieve the specified error string. The index runs from 1 to the value of <strong>NUM-MESSAGES</strong>.</td>
</tr>
<tr>
<td>GET-NUMBER( index ) method</td>
<td>Allows you to retrieve the specified error number. The index runs from 1 to the value of <strong>NUM-MESSAGES</strong>.</td>
</tr>
</tbody>
</table>

Testing for statement errors

The next example presumes that you want to execute special business logic if a particular system error message occurs. The **NO-ERROR** option suppresses the standard ABL error handling and the **IF** statement queries the ERROR-STATUS system handle message numbers to see if the particular error message occurred, as shown:

```
DEFINE VARIABLE i AS INTEGER NO-UNDO.
FIND FIRST Customer WHERE CustNum = 1000 NO-ERROR.
IF ERROR-STATUS:ERROR THEN
   DO i = 1 TO ERROR-STATUS:NUM-MESSAGES:
      IF ERROR-STATUS:GET-NUMBER( i ) = 565 THEN
         RUN FailedFindRecovery.p.
      IF i = ERROR-STATUS:NUM-MESSAGES THEN
         RUN GeneralErrorRecovery.p.
   END.
END.
```
Testing for built-in method warnings

Since built-in handle methods treat errors as warnings, you cannot use the IF ERROR-STATUS:ERROR test to detect them. However, even if the ERROR attribute is not set, the error messages are saved in the ERROR-STATUS object. Therefore, if your statement contains a method reference, use this test: IF ERROR-STATUS:NUM-MESSAGES > 0. For example:

```plaintext
DEFINE VARIABLE hSocket AS HANDLE.
CREATE SOCKET hSocket.
hSocket:CONNECT ("-H localhost -S 333") NO-ERROR.

IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
  RUN FailedSocketConnect.p.
```

Undoing work

The options of the UNDO statement and the ON ERROR phrase are nearly identical. The UNDO statement serves the role of undoing the work of the block when your business logic detects a problem that needs to be treated as an application error. Therefore, you need the same set of branching options to provide the same level of support for handling application errors.

Borrowing the test framework from Providing custom error handling on page 45 section, this example removes the explicit ON ERROR phrase and provides a nonsensical IF test that always leads to the execution of the UNDO statement:

```plaintext
PROCEDURE NestedBlocks:
  Outer-Block:
    FOR EACH Customer WHERE CustNum < 5:
      ASSIGN Customer.Name = "changed_" + Customer.Name.
  Inner-Block:
    FOR EACH Order OF Customer:
      DISPLAY OrderNum.
      /* Nonsense code tests for application error. */
      IF SUBSTRING(Customer.Name, 1, 8) EQ "changed_" THEN
        UNDO, RETURN ERROR.
      END.
  END.
END.

DISPLAY "For Blocks Complete".
END PROCEDURE.
RUN NestedBlocks.
DISPLAY "Procedure NestedBlocks Complete."
```
Raising the ERROR condition

You have already seen that you can raise ERROR with the RETURN ERROR option of both the ON ERROR phrase and the UNDO statement. The RETURN statement with the ERROR option is the third way to raise ERROR.

In general, the RETURN statement allows you to exit a block immediately. Immediately means that execution of the block halts with the RETURN statement and returns to the calling block without consideration to intervening blocks.

The statement leaves the local or remote procedure or user-defined function block, trigger block, database trigger block, the method block of a class, the class constructor block, or the property accessor block, and returns to the calling procedure, user-defined function, method, constructor, or property accessor. If there is no caller, RETURN returns to the tool that invoked the routine. This is the syntax for the RETURN statement:

Syntax

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

`return-value`

The value that RETURN returns to the caller with or without the ERROR condition.

`ERROR`

Causes an ERROR condition in the calling block. This can cause the ERROR condition to be raised for the following statements in the caller:

- The RUN statement for a procedure
- Any statement that invokes a method of a class
- Any statement that invokes the NEW or DYNAMIC-NEW function to instantiate a class (invoking the specified constructor and all other constructors for the class hierarchy)
- Any statement that accesses a property defined with property accessors

You can use the ERROR option in a procedure, database trigger block, class-based method, constructor, property accessor method. However, you cannot use the ERROR option in a user-interface trigger block or destructor to raise ERROR outside of the trigger block. Any values that are set for OUTPUT or INPUT-OUTPUT parameters before the RETURN ERROR executes are not returned to the caller.

The following table shows how to access return-value in the caller in various cases:
In this case . . . | How to retrieve the return value in the caller . . .
---|---
The *return-value* is specified without the *ERROR* option in a procedure or trigger block. | Access the `RETURN-VALUE` function.
The *return-value* is specified for a non-VOID method or user-defined function without the *ERROR* option. | Access the method or function return value by referencing the function or method call in an expression, similar to referencing a variable.
The *return-value* is specified with the *ERROR* option. | Access the `RETURN-VALUE` function. For structured error handling, you can also access the `ReturnValue` property of the error object.

`error-object-expression` *(structured error handling)*

This feature is documented in Using ABL Error Classes on page 69

**NO-APPLY**

Suppresses the default behavior for the current user-interface event. You can use the *NO-APPLY* option in a user-interface trigger block to suppress that behavior. For example, the default behavior for a character code key press in a fill-in field is to echo the character in the field. If you execute `RETURN NO-APPLY` in a trigger, this behavior is not performed and *NO-APPLY* returns without setting a *return-value*.

**User defined functions**

The user-defined function, which is defined by the `FUNCTION` statement, returns a value of a specific data type as its primary function. The `RETURN` statement is the tool you use to specify in the function body what value to return to the caller. This fact makes it impossible to use the `RETURN ERROR` statement in the same way as it is used in other blocks.

`RETURN ERROR` in a user-defined function does not raise error in the caller. Instead, it sets the target variable of the function to the unknown value. Therefore, you could perform error checking on a function call by checking for the unknown value after a function call. This technique only works if the function uses the `RETURN ERROR` statement and the target variable has a value other than the unknown value at the time of the function call. For example:

```plaintext
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.".
```
When run, this code outputs:

```
Error in user-defined function.
```
STOP and QUIT Condition Handling

For details, see the following topics:

- STOP condition handling
- QUIT condition handling

STOP condition handling

The STOP condition occurs when a STOP statement is executed or when a block statement contains the STOP-AFTER phrase and the time-limit is exceeded. The AVM also automatically raises the STOP condition when an unrecoverable system error occurs; for example, when a database connection is lost or a RUN statement cannot find a procedure file.

A user can raise the STOP condition by pressing the key mapped to the STOP key while input is enabled (or anytime in a character interface). This STOP key is usually mapped as follows:

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

Default handling

When the STOP condition occurs, by default, the AVM follows these steps:
1. If the STOP condition results from an unrecoverable system error, such as a RUN statement that specifies a non-existent procedure file, the AVM displays an error message to the standard output device, which is usually the terminal screen. The NO-ERROR option does not suppress error messages resulting from a STOP condition.

2. The AVM undoes the current transaction.

3. The AVM looks for an ON STOP phrase. If the AVM finds an ON STOP phrase, it must determine if it is safe to resume execution at this point. In some cases, the AVM might ignore ON STOP phrases at certain levels of the call stack. For example, if the AVM executes a procedure that relies on a lost database connection, the AVM raises the STOP condition and unwinds the call stack until it gets to a level above all references to the lost database. If it encounters an ON STOP before this point it ignores it. If it encounters an ON STOP phrase after this point, then the AVM executes the ON STOP.

4. If the application was started from a tool such as the OpenEdge Editor, it returns to that tool. Otherwise, if you used the Startup Procedure (-p) parameter to start the ABL session, and if the startup procedure is still active, the AVM restarts the startup procedure.

Example: default handling for lost database connections

If OpenEdge loses a database connection (for example, because a remote server failed), client processing might continue. In this case, the AVM:

- Raises the STOP condition. The AVM ignores ON STOP phrases.
- Deletes any persistent procedures that reference the disconnected database.
- Undoes blocks beginning with the innermost active block and working outward. It continues to undo blocks until it reaches a level above all references to tables or sequences in the lost database.
- At this point, the AVM will honor any further ON STOP phrases you have coded.
- Continues to undo blocks until it reaches an ON STOP phrase. If no ON STOP phrase is reached, it undoes all active blocks and restarts the top level procedure.

Raising the STOP condition

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

Syntax

```
STOP
```
Overriding default handling

The default handling for the STOP condition is almost always appropriate. However, you can override the default handling by adding the ON STOP phrase to a REPEAT, FOR, or DO statement. The example below demonstrates the ON STOP phrase:

```
DO ON STOP UNDO, RETRY:
  IF RETRY THEN DO:
    DISPLAY "Application encountered a STOP condition and will terminate.";
    UNDO, LEAVE.
  END.
ELSE RUN MissingCode.p.END.
DISPLAY "Application continues...". /* Never executes. */
```

In this example, the ON STOP phrase uses the RETRY branching option to provide an opportunity to recover from the STOP condition. The RETRY function in the IF statement allows you to determine if you are in RETRY mode.

This example uses the RETRY =TRUE branch of the IF statement to display another error message. This safe structure might be used to allow an application user to help locate a missing file or to ask the application user about whether or not to save data before terminating.

Here is the complete syntax for the ON STOP phrase:

**Syntax**

```
ON STOP 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`, ON STOP UNDO undoes the processing of the block started by the statement that contains the ON STOP 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`.
NEXT \([\text{label2}]\)

Indicates that after undoing the processing of a block, the AVM executes the next iteration of the block you name with the \text{label2} option. If you do not name a block with the NEXT option, the AVM executes the next iteration of the block labeled with \text{label1}.

RETRY \([\text{label1}]\)

Indicates that after undoing the processing of a block, the AVM repeats the same iteration of the block you name with the \text{label1} option.

RETRY is the default processing if you do not use LEAVE, NEXT, RETRY, or RETURN.

RETURN ...

Returns to the calling routine, or if there is no calling routine, returns to the OpenEdge Editor. The following table describes various RETURN cases:

Table 6: RETURN cases

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{return-value}</td>
<td>The CHARACTER string you provide is passed to the caller. The caller can use the \text{RETURN-VALUE} function to read the returned value. For user-defined functions, the value must match the specified return type.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Raises \text{ERROR} in the caller and undoes the current subtransaction. You cannot specify \text{ERROR} within a user-interface trigger block or a destructor.</td>
</tr>
<tr>
<td>ERROR \text{return-value}</td>
<td>Raises \text{ERROR} in the caller and undoes the current subtransaction (except for user-defined functions). The CHARACTER string you provide is available to the caller in the \text{RETURN-VALUE} function. In structured error handling, the AVM also creates an \text{AppError} object and stores the \text{return-value} in the \text{ReturnValue} property.</td>
</tr>
<tr>
<td>ERROR \text{error-object-expression}</td>
<td>In structured error handling, raises \text{ERROR} in the caller and undoes the current subtransaction. The specified error object is your code. If it is an \text{AppError} object, the caller can also use the \text{RETURN-VALUE} function to read the setting of the \text{ReturnValue} property.</td>
</tr>
<tr>
<td>NO-APPLY</td>
<td>In a user-interface trigger, prevents the AVM from performing the default behavior for that event.</td>
</tr>
</tbody>
</table>
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:

**Syntax**

```
STOP-AFTER expression
```

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 and default STOP condition handling occurs. Use an ON STOP phrase on the block (or an enclosing block) to alter the default STOP condition handling.

If the block iteration completes before the specified time expires, the timer resets to expression for the next iteration. In other words, the timer is limited to the scope of a single block iteration. If a block with a STOP-AFTER phrase encloses another block or calls another block, the timer continues while the inner blocks execute.

If a block with a STOP-AFTER phrase contains a nested block with a STOP-AFTER phrase, then each has a timer in effect. If the outer block timer expires while the inner block is executing, the STOP condition is raised even if the timer for the inner block has not expired.

If the STOP condition is handled and execution resumes within the scope of a block with a STOP-AFTER phrase, no timer is in effect until the next iteration of a block with a STOP-AFTER phrase. In other words, all old timers are dismissed but new timers can now be established.

When the timer expires, the STOP condition is raised on the current statement.

Two important use cases for the STOP-AFTER phrase are to time-limit dynamic queries and to time-limit a procedure call. The following example time-limits a procedure call using a RUN statement:

```
DEFINE VARIABLE cnt as INTEGER INITIAL 0.
PROCEDURE bumpCnt:
    cnt = cnt + 1.
END.

DO STOP-AFTER 5:
    RUN bumpCnt.
END.
```

Use this technique to also make timed calls to class methods and user-defined functions.
The following example is simplified code that lets you try different STOP-AFTER cases.

```plaintext
DEFINE VARIABLE EndlessCount AS INTEGER INITIAL 0.
DO STOP-AFTER 5 ON STOP UNDO, LEAVE:
    FOR EACH Customer STOP-AFTER 1:
        ASSIGN EndlessCount = EndlessCount + 1.
        /* Try a complex operation on a Customer record to use up the timer
           in a single iteration and raise the STOP condition in the inner
           block */
    END.
MESSAGE "Procedure half complete. Endlesscount = " EndlessCount ".".
REPEAT STOP-AFTER 1:
    ASSIGN EndlessCount = EndlessCount + 1.
    /*IF EndlessCount > 2000 THEN LEAVE. */
    END.
MESSAGE "Procedure nearly complete. Endlesscount = " EndlessCount ".".
END.
MESSAGE "Procedure complete. Endlesscount = " EndlessCount "."
```

If you run this code as is, the outer DO block establishes a 5 second time limit for the work of the DO block and all inner blocks. When the inner FOR EACH block starts, another timer is established for the first iteration of this block. When the first FOR EACH iteration completes, its timer is reset to 1 second for the next iteration. Meanwhile, the outer timer on the DO block continues without interruption.

The FOR EACH block completes and execution continues forward to the REPEAT block, which is an endless loop. The REPEAT block also has a 1 second timer for each iteration of the block. At some point, the outer 5 second timer elapses and the AVM raises the STOP condition. The STOP condition is raised on the statement the AVM was executing when the timer elapsed. Normal STOP handling proceeds from that point.

As the stack unwinds during STOP processing, the AVM encounters the ON STOP phrase on the DO block. The ON STOP phrase dismisses the STOP condition and resumes normal execution with the next statement following the DO block, as directed by the LEAVE option.

If you remove the comments from the IF statement in the REPEAT block, the block will complete within the outer time limit and the STOP condition is not raised.

If you want to experiment with elapsed timers on an inner block, insert a complex operation inside the FOR EACH block.
In the following example, the STOP-AFTER expression is modified during program execution:

```plaintext
DEFINE VARIABLE ix AS INTEGER NO-UNDO.
DEFINE VARIABLE stopTime AS INTEGER NO-UNDO INITIAL 30.
DO WHILE TRUE STOP-AFTER stopTime ON STOP UNDO, LEAVE:
    RUN spinHere (10000).
    stopTime = stopTime / 2.
END.
MESSAGE "program finished".

PROCEDURE spinHere:
    DEFINE INPUT PARAMETER spinLimit AS INT64 NO-UNDO.
    DEFINE VARIABLE endTime AS INT64 NO-UNDO.
    DEFINE VARIABLE loopFlag AS LOGICAL NO-UNDO.
    ASSIGN
        loopFlag = TRUE
        endTime = ETIME(FALSE) + spinLimit.
    DO WHILE loopFlag:
        IF (ETIME(FALSE) > endTime) THEN
            loopFlag = FALSE.
        END.
    END.
END PROCEDURE.

Because the STOP-AFTER expression is re-evaluated for each iteration of a looping block, any changes made to the expression during the iteration effect the timer for the block. In the example, the STOP-AFTER time limit is specified by the variable stopTime, which is initially set to 30 seconds. The procedure contains an iterating block which runs a procedure that executes for 10 seconds.

On the first iteration of the DO WHILE TRUE loop, stopTime is 30 seconds. The loop executes for 10 seconds, and then divides stopTime by 2. On the second iteration, the stopTime is 15 seconds; again the loop executes for 10 seconds, and then divides stopTime by 2. On the third iteration, the stopTime is 8 seconds. This time, the procedure spinHere runs for 8 seconds and then raises STOP. The STOP condition is handled by the DO block, and then the program displays the message program finished.

If a code block is called with a time limit of zero, the block is executed as if the STOP-AFTER phrase was omitted from the block declaration.

Consider the following example:

```plaintext
DEFINE VARIABLE barLimit AS INTEGER NO-UNDO.
DEFINE VARIABLE ix AS INTEGER NO-UNDO INITIAL 1.
DO STOP-AFTER 10 ON STOP UNDO, LEAVE:
    RUN foo.
END.

PROCEDURE foo:
    RUN bar.
END PROCEDURE.

PROCEDURE bar:
    DO WHILE ix > 0 STOP-AFTER barLimit:
        ix = ix + 1.
    END.
END PROCEDURE.
```
In this example, procedure foo is run from within a timed block with a 10 second time limit; procedure bar is called from within the timed block, and contains an iterating block that specifies the STOP-AFTER phrase. Because the value of the STOP-AFTER expression evaluates to zero (that is, the current value of the barLimit variable), the block within bar is executed as an untimed block. However, the rules for execution of an untimed block within a timed block apply, so the untimed block in bar is executed with an implicit iteration time limit of 10 seconds.

Other points to consider are:

- If the expression evaluates to zero or less, then this is the equivalent of not specifying a STOP-AFTER phrase.
- STOP-AFTER phrases are not intended to interact with user interfaces.
- Blocking calls to third party software components, where the AVM has transferred execution control, cannot be timed out. This category includes operating system calls, MS Windows system calls, and calls to any third party DLLs and Unix shared objects.

QUIT condition handling

ABL supports a QUIT statement to terminate an application completely. The AVM raises the QUIT condition only when it encounters a QUIT statement.

Default handling

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

1. Commits the current transaction.

2. Exits the ABL session. If the ABL session is running on the AppServer, terminating it causing the AppServer server to shut down. The AVM then returns to the ABL client session from which it was spawned.

3. 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.

Note: Even if the -p startup option is in effect (to define the main procedure in your application), the AVM returns to the operating system.

Raising the QUIT condition

The QUIT statement raises the QUIT condition and directs the enclosing block to perform its default QUIT condition handling. This is the statement syntax:

**Syntax**

QUIT
Overriding default handling

The default handling of the QUIT condition is almost always appropriate. However, by using the ON QUIT phrase in a REPEAT, FOR EACH, or DO block you can modify the default QUIT condition handling.

This example uses the ON QUIT phrase to provide a branch within the block that can be executed before the application quits:

```plaintext
DO ON QUIT UNDO, RETRY:
  IF RETRY THEN DO:
    DISPLAY "Application encountered a QUIT condition and will terminate.".
    UNDO, LEAVE.
  END.
FIND FIRST customer WHERE CustNum = 1000 NO-ERROR.
  IF ERROR-STATUS:ERROR THEN QUIT.
END.
DISPLAY "Application continues...". /* Never executes. */
```

The RETRY function checks whether the block is in RETRY mode and displays an error message. You can easily replace the DISPLAY statement that performs tasks necessary to ensure an orderly shutdown of the application.

Here is the complete syntax for the ON QUIT phrase:

**Syntax**

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

UNDO [label1]
```

Indicates that the specified block is undone. If you do not specify the UNDO option, then the current transaction is committed when the QUIT statement is executed.
LEAVE \[label2\]

Indicates that after committing or undoing the transaction, the AVM leaves the block labeled \textit{label}. If you do not name a block, the AVM leaves the block with the \texttt{ON QUIT} phrase in its heading.

NEXT \[label2\]

Indicates that after committing or undoing the transaction, the AVM executes the next iteration of the block you name with the \textit{label} option. If you do not name a block with the \texttt{NEXT} option, the AVM executes the next iteration of the block with the \texttt{ON QUIT} phrase in its heading.

RETRY \[label1\]

Indicates that after committing or undoing the processing of a block, the AVM repeats the same iteration of the block that was undone or committed.

\texttt{RETRY} is the default if you do not specify \texttt{LEAVE}, \texttt{NEXT}, \texttt{RETRY}, or \texttt{RETURN}.

RETURN ...

Returns to the calling routine, or if there is no calling routine, returns to the OpenEdge Editor. The following table describes various \texttt{RETURN} cases:

\textbf{Table 7: RETURN cases}

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{return-value}</td>
<td>The \texttt{CHARACTER} string you provide is passed to the caller. The caller can use the \texttt{RETURN-VALUE} function to read the returned value. For user-defined functions, the value must match the specified return type.</td>
</tr>
<tr>
<td>\texttt{ERROR}</td>
<td>Raises \texttt{ERROR} in the caller and undoes the current subtransaction. You cannot specify \texttt{ERROR} within a user-interface trigger block or a destructor.</td>
</tr>
<tr>
<td>\texttt{ERROR return-value}</td>
<td>Raises \texttt{ERROR} in the caller and undoes the current subtransaction (except for user-defined functions). The \texttt{CHARACTER} string you provide is available to the caller in the \texttt{RETURN-VALUE} function. In structured error handling, the AVM also creates an \texttt{AppError} object and stores the \texttt{return-value} in the \texttt{ReturnValue} property.</td>
</tr>
</tbody>
</table>
## QUIT condition handling

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR error-object-expression</td>
<td>In structured error handling, raises ERROR in the caller and undoes the current subtransaction. The specified error object is your code. If it is an AppError object, the caller can also use the RETURN-VALUE function to read the setting of the ReturnValue property.</td>
</tr>
<tr>
<td>NO-APPLY</td>
<td>In a user-interface trigger, prevents the AVM from performing the default behavior for that event.</td>
</tr>
</tbody>
</table>
Using Structured Error Handling

Introduction to Condition Handling on page 19 introduced the idea of structured error handling and provided a simple example of the mechanics of the model within ABL. This chapter describes all the components of the model while demonstrating the overlap and the differences with the traditional error handling model.

For details, see the following topics:

- Comparison to traditional error handling
- Additional language features
- Structured error handling tasks
- Next steps

Comparison to traditional error handling

The remainder of this manual will help you build your structured error handling skills with a thorough explanation of each component of the model and the associated programming tasks. It is useful to review traditional error handling, as it forms the roots of structured error handling.

Your knowledge of traditional error handling is an essential beginning for understanding structured error handling in ABL. Traditional Error Handling on page 37 began with a list of tasks you needed to understand to successfully use traditional error handling. The following table presents a related similar list of traditional error handling concepts and summarizes how the particular concept changes in structured error handling.
<table>
<thead>
<tr>
<th>Traditional error handling concept</th>
<th>Comparison to structured error handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know what raises <strong>ERROR</strong></td>
<td>Whether a statement raises <strong>ERROR</strong> is not normally dependent on which error handling model is in effect in a block. However, the presence of a <strong>CATCH</strong> block in any block allows all errors to be handled more consistently, since it can handle errors that cannot be handled by the <strong>NO-ERROR</strong> option or <strong>ON ERROR</strong> phrase. In particular, statements that do not raise <strong>ERROR</strong> in traditional error handling, like the <strong>MESSAGE</strong> statement, now raise <strong>ERROR</strong> when <strong>CATCH</strong> is present. In traditional error handling, methods on built-in system handles treat errors as warnings. Therefore, built-in methods do not raise <strong>ERROR</strong>. If a <strong>CATCH</strong> block is present in the block containing a failed built-in method, the AVM will now raise <strong>ERROR</strong>. Note that an error handled by a <strong>CATCH</strong> block makes the block subject to <strong>UNDO</strong>. The <strong>NO-ERROR</strong> option is still useful to suppress an error and avoid <strong>UNDO</strong> of the block.</td>
</tr>
<tr>
<td>Understanding OpenEdge messages</td>
<td>System error messages are the same. Because they are wrapped by error objects, you access the messages by properties and methods on the error object, as opposed to attributes and methods on the <strong>ERROR-STATUS</strong> system handle.</td>
</tr>
<tr>
<td>Know how a particular block handles an error</td>
<td>Default error handling is the same in both models.</td>
</tr>
<tr>
<td>Know how to use block statement options to alter default error handling</td>
<td>For explicit error handling, the <strong>ON ERROR</strong> phrase and <strong>UNDO</strong> statement all support the <strong>THROW</strong> option. <strong>THROW</strong> suppresses display of default error messages, exits the block, and passes the error raised in the block to the immediate outer block. Two statements enable you to change default error handling behavior for whole files: <strong>ROUTINE-LEVEL ON ERROR UNDO</strong>, <strong>THROW</strong> and <strong>BLOCK-LEVEL ON ERROR UNDO</strong>, <strong>THROW</strong>. Similarly, a compile-time startup parameter, <strong>-undothrow</strong>, lets you implement the same behaviors for an entire application.</td>
</tr>
<tr>
<td>Know how to suppress unwanted errors</td>
<td>When a <strong>CATCH</strong> block for a particular error type is in effect, the AVM will not display an error message. The error message is instead placed within the error object created by the AVM. If there is no <strong>CATCH</strong> block for the particular error type, then the AVM will display an error to the standard output. The <strong>NO-ERROR</strong> option is an important part of structured error handling. It is the only option that can prevent a statement within a block from raising <strong>ERROR</strong> where it otherwise would. Even when a <strong>CATCH</strong> block is present that would handle the error a statement raises, if the <strong>NO-ERROR</strong> option is used on the statement, the <strong>CATCH</strong> block ignores it.</td>
</tr>
</tbody>
</table>
### Additional language features

ABL structured error handling uses the following language elements:

- An extensible hierarchy of classes to allow OpenEdge system errors and your custom application errors to be expressed as objects.

- The `UNDO, THROW` directive with the `ON ERROR` phrase to allow you to throw unhandled (uncaught) errors to the next outer block in the call stack. This feature also allows you to easily throw errors outside of a block. This is a significant enhancement to the flexibility of ABL error handling.

- The `THROW` option to the `UNDO` statement. This option is your tool for generating and throwing custom application errors. (You cannot explicitly generate an OpenEdge system error, but you can re-throw one with this option.)

- Two statements that change default behavior in the containing file from traditional to structured error handling. The first of these statements, `BLOCK-LEVEL ON ERROR UNDO, THROW`, causes all blocks that have no explicit error directive and are subject to default error handling to use the `UNDO, THROW` directive (replacing `UNDO, RETRY` or `UNDO, LEAVE`). The second statement, `ROUTINE-LEVEL ON ERROR UNDO, THROW`, has the same effect on routine-level blocks but does not change the default behavior for other blocks. See THROW as a default for blocks on page 99.

- The compile-time startup parameter `-undothrow n` to enable you to implement the behavior associated with `BLOCK-LEVEL ON ERROR UNDO, THROW` or `ROUTINE-LEVEL ON ERROR UNDO, THROW` for an entire application, rather than on a per-file basis. See the OpenEdge Deployment: Startup Command and Parameter Reference for more information on `-undothrow`.

---

### Traditional error handling concept vs. Comparison to structured error handling

<table>
<thead>
<tr>
<th>Traditional error handling concept</th>
<th>Comparison to structured error handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know how to test for errors after a NO-ERROR option and branch to your custom error handling logic</td>
<td>If you use the NO-ERROR option to prevent a statement from being handled by a CATCH block, then the ERROR-STATUS system handle continues to be the tool you use to determine if an error occurred on the statement and what the error or errors were. You can think of the NO-ERROR and ERROR-STATUS combination as a miniature CATCH block on a single statement.</td>
</tr>
<tr>
<td>Know how to raise ERROR and specify an application custom error data</td>
<td>The RETURN ERROR statement and RETURN option of the ON ERROR phrase and UNDO statement are still your ways for raising application errors. You can now raise (THROW) your own error object. The THROW option also provides the way to raise ERROR in a user-defined function, as explained in detail in THROW with user-defined functions on page 104.</td>
</tr>
<tr>
<td>Know how to undo transactions</td>
<td>The UNDO statement is still your method for undoing transactions. Structured error handling adds the THROW option to this statement.</td>
</tr>
<tr>
<td>Understanding STOP and QUIT condition processing</td>
<td>Structured error handling does not handle STOP and QUIT conditions.</td>
</tr>
</tbody>
</table>
• A **CATCH** statement, used to define an error-handling end block that is associated with an error type and any undoable ABL block.

• A new end block with the **FINALLY** statement. The **FINALLY** statement lets you define an end block that must execute at the end of an ABL block (or each iteration) whether the block executed successfully or raised an error. The **FINALLY** block also executes after a **CATCH** block, if one exists.

• Class-based error objects can be thrown from an AppServer to an ABL client. For the criteria an error object must satisfy to be thrown from server to client, see **Throwing class-based error objects from an AppServer to an ABL client** on page 71.

## Structured error handling tasks

Another way to introduce structured error handling is to look at a list of additional tasks you need to know how to do. The following table expands on the list presented in **Traditional versus structured error handling table**.

### Table 9: Structured error handling tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know how to reference an error object and access its properties and methods</td>
<td>Define an instance variable and type (class name) in the <strong>CATCH</strong> block header. Use the instance variable to access the properties and methods.</td>
</tr>
<tr>
<td>Know how to subclass the built-in application error class (optional)</td>
<td>Structured error handling lets you define your own application error objects using the full set of object-oriented programming features. Application error objects must be a subclass of <strong>Progress.Lang.AppError</strong>. This task requires a working knowledge of object-oriented programming in ABL.</td>
</tr>
<tr>
<td>Know how to explicitly raise an error (<strong>THROW</strong>)</td>
<td>The <strong>THROW</strong> option is another branching directive like <strong>NEXT</strong> or <strong>LEAVE</strong>. It is available in all the places that the other directives are available.</td>
</tr>
<tr>
<td>Know how to handle a particular error and how to handle different errors within a block</td>
<td>Each error type you need to handle differently in a block requires a different <strong>CATCH</strong> block. Since the first compatible <strong>CATCH</strong> block found is used, organize <strong>CATCH</strong> blocks from most specific to least specific.</td>
</tr>
<tr>
<td>Know how to pass (re-<strong>THROW</strong>) error objects from an inner block to an outer block</td>
<td>Once you catch an error object, you can use the <strong>THROW</strong> directive to pass it to the enclosing block. That block can re-throw the error to the next enclosing block, and so on.</td>
</tr>
<tr>
<td>Know how to propagate errors up the call stack automatically</td>
<td>The <strong>BLOCK-LEVEL ON ERROR UNDO</strong>, <strong>THROW</strong> and <strong>ROUTINE-LEVEL ON ERROR UNDO</strong>, <strong>THROW</strong> statements and the <strong>-undothrow</strong> startup parameter let you make <strong>THROW</strong> the default directive on a per-file or per-application basis, facilitating centralized error handling by propagating errors upward.</td>
</tr>
</tbody>
</table>
When an error occurs, the AVM looks for an error handler using this precedence, from highest to lowest:

- Statement NO-ERROR option
- CATCH block
- Explicit ON ERROR phrase
- Implicit ON ERROR phrase

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

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

Next steps

The rest of this manual continues to build your structured error handling skills. Each chapter has a specific ABL element as its topic, but in many cases, simple examples of the element are introduced before the main documentation for it. By the end of the manual, you will have specific knowledge of all of the ABL components of error handling, and you will also understand the interactions between the elements.

The remaining chapters explain:

- The hierarchy of ABL error classes and how these classes relate to specific use cases for CATCH blocks
- All about CATCH blocks with detailed code examples
- How to use the THROW directive
- How to use FINALLY blocks
Using ABL Error Classes

The first characteristic of structured error handling is that errors are represented as objects. This chapter introduces the interface and four classes that underpin the ABL error objects. You do not need to commit to object-oriented application design to use structured error handling. The simple mechanics of structured error handling are strongly related to the mechanics of traditional error handling.

At the same time, structured error handling is an object-oriented programming technique and you can create your own error types. If you want to use structured error handling at this level, then you should be familiar with ABL object-oriented programming before beginning your design work. (See *OpenEdge Development: Object-oriented Programming* for more information.)

For details, see the following topics:

- Progress.Lang.Error interface
- Progress.Lang.ProError class
- Progress.Lang.SysError class
- Progress.Lang.SoapFaultError class
- Progress.Lang.AppError class
- 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 to interact with the ABL structured error handling model. This interface cannot be implemented by a user-defined class. Instead, create a subclass of the Progress.Lang.AppError class to create your own type of ABL error object.

The interface defines the properties and methods shown in the following table:

Table 10: Properties and methods

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallStack property</td>
<td>Returns a string representing the call stack at the time the error object was thrown. 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 session startup parameter.</td>
</tr>
<tr>
<td>NumMessages property</td>
<td>In ABL, an error is represented as a pair of values. The message number is a unique number identifying the particular error. The error message is a string which describes the error. This property indicates how many error numbers and error messages the error object contains.</td>
</tr>
<tr>
<td>Severity property</td>
<td>The Severity property is not used by ABL system errors and returns zero if accessed. It is provided as a mechanism for you to use to assign severity rankings to your various application errors (Progress.Lang.AppError).</td>
</tr>
<tr>
<td>GetMessage(MessageIndex)</td>
<td>Returns the error message for the indexed error in the error object, beginning with one (1). If there is no error message at the indicated index, the method returns the empty string.</td>
</tr>
<tr>
<td>GetMessageNum(MessageIndex)</td>
<td>Returns the error message number associated with the indexed error in the error object. For Progress.Lang.SysError and Progress.Lang.SoapFaultError objects, the method returns the message number for the system generated error. If there is no error message at the index, the method returns the empty string.</td>
</tr>
</tbody>
</table>
Specifying `Progress.Lang.Error` interface in a `CATCH` statement creates an error handler that catches all possible errors. For example:

```
DO ON ERROR UNDO, THROW:
  .
  .
  .
  CATCH anyErrorObject AS Progress.Lang.Error:
    .
    .
  END CATCH.
END. /* DO */
```

### Throwing class-based error objects from an AppServer to an ABL client

Error objects can be thrown from an AppServer to an ABL client if their classes implement the `Progress.Lang.Error` interface and they satisfy additional conditions, given below. Error objects passed from AppServer to client are serialized on the sending side and then deserialized on the receiving side. The following restrictions apply to error objects thrown from an AppServer to a client:

- In the case of a user-defined class, the object’s class and all of the classes in its hierarchy must be marked as `SERIALIZABLE`. For more information on marking a class as `SERIALIZABLE`, see the `CLASS` statement entry in *OpenEdge Development: ABL Reference*.
- The content of static data members is not serialized, and the state of queries, buffers, open files, streams, and event subscriptions, for example, are not maintained.
- All of the object’s data members that are defined as class-based objects must be of a class type that is also marked `SERIALIZABLE`. (This restriction does not apply to static data members that are defined as objects, because static data members are not serialized.)
- Handle-based variables are serialized, but no information for reconstructing handle-based objects on the receiving side is serialized.
- `MEMPTRs` assigned by an ABL application are serialized, but `MEMPTRs` from an external source (such as a DLL or shared library) are not serialized.
- Statically defined temp-tables and ProDataSets in user-defined classes are serialized, except for `REFERENCE-ONLY` tables.
- The `REJECTED`, `ERROR`, `ERROR-STRING`, and `DATA-SOURCE-MODIFIED` attributes for temp-tables in an object are maintained as part of the deserialization process. Similarly, the `REJECTED` and `ERROR` attributes for ProDataSets are maintained during serialization.
- Not all built-in classes are serializable. See the `CLASS` statement entry in *OpenEdge Development: ABL Reference* for a full list of serializable built-in classes.
- .NET and ABL-extended .NET error objects cannot be thrown.
Progress.Lang.ProError class

Progress.Lang.ProError is the ultimate super class for all ABL built-in and user-defined classes that represent errors in the ABL structured error handling model. You cannot directly inherit from this class, 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 needed to interface with the ABL structured error handling model, as shown in Properties and methods table. This class provides the functionality to retrieve error messages, error numbers, and the contents of the error call stack.

A CATCH block that references this type handles all built-in and user defined errors, as shown:

```
DO ON ERROR UNDO, THROW:
   .
   .
   .
   CATCH anySystemOrAppErrorObject AS Progress.Lang.ProError:
      .
      .
      .
   END CATCH.
END. /* DO */
```

Progress.Lang.SysError class

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

Properties and methods table describes the properties and methods inherited by this class. Objects of SysError type can be thrown from an AppServer to an ABL client.

A CATCH block that references this type handles all system errors, as shown:

```
DO ON ERROR UNDO, THROW:
   .
   .
   .
   CATCH anySystemErrorObject AS Progress.Lang.SysError:
      .
      .
      .
   END CATCH.
END. /* DO */
```
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 and is a FINAL class.

SoapFaultError is a type of system error, and therefore you cannot instantiate the object with the NEW or DYNAMIC NEW function or create a user-defined class that inherits from it. The class constructors are reserved for system use only.

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

Table 11: SoapFaultError properties

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoapFault property</td>
<td>In traditional error handling, SOAP-fault information is available from the ERROR-OBJECT-DETAIL property of the ERROR-STATUS handle after a Web service call is invoked with the NO-ERROR option. In structured error handling, you can access the same information using a CATCH statement for the Progress.Lang.SoapFaultError object. The SoapFault property in this object contains the handle to the SOAP-fault object. The Soap-Fault-Detail property of the system handle provides the full detail about the original SOAP fault.</td>
</tr>
</tbody>
</table>

See the chapter on error handling in OpenEdge Development: Web Services for more detailed information on handling SOAP faults.

Caution: Like other error objects, objects of SoapFaultError type 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. For more information on serialization rules for class-based objects, see Throwing class-based error objects from an AppServer to an ABL client on page 71.
A CATCH block that references this type handles all SoapFaultError objects. Since SoapFaultError is a subclass of SysError, you may need two CATCH blocks to handle both general errors and SOAP faults. If this is the case, the more specific error type must come before the general type, as shown in the example below. The AVM executes the first compatible CATCH block it encounters.

```abl
DO ON ERROR UNDO, THROW:
  .
  .
  CATCH anySoapFaultErrorObject AS Progress.Lang.SoapFaultError:
    .
    .
    END CATCH.
  CATCH anySystemErrorObject AS Progress.Lang.SysError:
    .
    .
    END CATCH.
END. /* DO */
```

**Progress.Lang.AppError class**

Progress.Lang.AppError is the ultimate super class of all application errors. An application error is simply any collection of data you need to provide necessary information about a condition. Representing a user-defined error as an error object allows your application to throw and catch or return the error in the ABL structured error handling model.

The following figure represents the hierarchy of ABL classes that provide Progress.Lang.AppError with all its features:

**Figure 1: Hierarchy of ABL error classes**
Progress.Lang.ProError is the ultimate super class of all error objects in ABL. From Progress.Lang.Object it inherits the basic features of an ABL class. It also implements the Progress.Lang.Error interface, which provides the basic properties and methods for handling errors. Progress.Lang.AppError adds the ability to add your own error messages, and it also includes a ReturnValue property. You can also create your own hierarchy of more complex AppError types by creating subclasses of AppError.

In traditional error handling, a called context can communicate with a calling context by using the RETURN ERROR statement and optionally providing a descriptive CHARACTER string. This returned string is analogous to an application error since it amounts to a user-defined error message or error data. You use the RETURN-VALUE function in the caller to access the string. When the AVM encounters the RETURN ERROR statement, it throws a Progress.Lang.AppError error object and places any specified string in the object’s ReturnValue property. The error object itself is only seen if it is caught by a CATCH block.

Progress.Lang.AppError class provides the ability to define error objects at run time. In ABL, user-defined error types are called application errors. The Progress.Lang.AppError class lets you define application errors that are handled like ABL system errors. In this paradigm, an error object consists of one or more descriptive CHARACTER messages, each with an associated INTEGER message number.

Application errors can be thrown from an AppServer to an ABL client. See Throwing class-based error objects from an AppServer to an ABL client on page 71 for a full list of restrictions on throwing error objects from an AppServer to a client.

Properties and methods

Properties and methods table describes the properties and methods inherited by this class. The following table describes the additional properties and methods of AppError.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReturnValue property</td>
<td>An AppError object can be used to replace RETURN ERROR error-data-string statements used in traditional error handling. In traditional error handling, you can use the RETURN-VALUE function in the caller to access the error-data-string. The ReturnValue property provides parallel functionality when you use a RETURN ERROR AppError-object-expression statement in structured error handling. The error-data-string is available in both the RETURN-VALUE function and the ReturnValue property in the caller. In fact, when the AVM detects RETURN ERROR error-data-string statement, it creates an AppError object and populates the ReturnValue property so that the caller can handle the RETURN ERROR with a CATCH block.</td>
</tr>
<tr>
<td>Severity property</td>
<td>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 applications.</td>
</tr>
</tbody>
</table>
### Constructors

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

**Syntax**

```plaintext
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. This error message and message number can be accessed with the `GetMessage(1)` and `GetMessageNum(1)` methods. This is the syntax:

**Syntax**

```plaintext
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. This is the syntax:

**Syntax**

```plaintext
PUBLIC AppError( INPUTReturnValue AS CHARACTER )
```
**Syntax**

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

**Example**

This section provides the class file of an AppError subclass and a procedure that demonstrates how to use the AppError methods.

This is the code for the class file (.cls):

```
CLASS MyAppError2 INHERITS Progress.Lang.AppError:
  DEFINE VARIABLE lastmsg AS INTEGER NO-UNDO.
  CONSTRUCTOR PUBLIC MyAppError2( ):
    SUPER().
  END CONSTRUCTOR.
  CONSTRUCTOR PUBLIC MyAppError2(mymsg AS CHARACTER, mynum AS INTEGER):
    AddMessage(mymsg, mynum).
    MESSAGE "In MyAppError2 Char & Int Constructor" mymsg mynum.
  END CONSTRUCTOR.
  CONSTRUCTOR PUBLIC MyAppError2(mynum AS INTEGER):
    MESSAGE "In MyAppError2 Integer Constructor" mynum.
  END.
  DESTRUCTOR PUBLIC MyAppError2():
    MESSAGE "In MyAppError2 destructor".
  END DESTRUCTOR.
END CLASS.
```

Notice the constructors provide messages. When an instance of the new class is created, these messages will appear in the run-time message area of the procedure window.

This is the code for the procedure:

```
/* ************** Definitions ************** */
DEFINE VARIABLE i AS INTEGER NO-UNDO.
DEFINE VARIABLE myaerr AS CLASS MyAppError2 NO-UNDO.
DEFINE VARIABLE mynm AS INTEGER NO-UNDO.
/* ************** Main Block ************** */
DO ON ERROR UNDO, LEAVE:
  RUN intproce.
  CATCH mae AS MyAppError2:
    MESSAGE "Inside MyAppError2 Catch".
    ASSIGN mynm = mae:NumMessages.
    MESSAGE "Number of messages" mynm .
    REPEAT i = 1 TO mae:NumMessages:
      MESSAGE "Error Number: " mae:GetMessageNum(i) SKIP
      "Message: " mae:GetMessage(i) SKIP
      "NumMessage: " i .
  END.
END.
```
MESSAGE "Here is the ReturnValue " mae:ReturnValue .
END CATCH.
END.

DO ON ERROR UNDO, LEAVE:
  RUN removeall.

CATCH mae AS MyAppError2:
  MESSAGE "Inside MyAppError2 Catch".
  ASSIGN mynm = mae:NumMessages.
  MESSAGE "Number of messages" mynm .
  REPEAT i = 1 TO mae:NumMessages:
    MESSAGE "Error Number: " mae:GetMessageNum(i) SKIP
      "Message: " mae:GetMessage(i) SKIP
      "NumMessage: " i .
  END.

  MESSAGE "Here is the ReturnValue " mae:ReturnValue .
END CATCH.
END.

/* Internal Procedures */
PROCEDURE intproce:
  MESSAGE "In intproce" .
  myaerr = NEW MyAppError2("ConstMsg", 100).
  myaerr:AddMessage("ThirdMsg", 103).
  REPEAT i = 1 TO myaerr:NumMessages:
    MESSAGE "Error Number: " myaerr:GetMessageNum(i) SKIP
      "Message: " myaerr:GetMessage(i) SKIP
      "NumMessage: " i .
  END.
  myaerr:RemoveMessage(2).
  myaerr:RemoveMessage(3).
  myaerr:ReturnValue = "Add and Remove Msg case 66".
  RETURN ERROR myaerr.
END.

PROCEDURE removeall.
  MESSAGE "In Removeall" .
  myaerr = NEW MyAppError2("ConstMsg", 200).
  myaerr:AddMessage("FourthMsg", 204).
  REPEAT i = 1 TO myaerr:NumMessages:
    MESSAGE "Error Number: " myaerr:GetMessageNum(i) SKIP
      "Message: " myaerr:GetMessage(i) SKIP
      "NumMessage: " i .
  END.
  myaerr:RemoveMessage(5).
  myaerr:RemoveMessage(4).
  myaerr:RemoveMessage(3).
  myaerr:RemoveMessage(2).
  myaerr:RemoveMessage(1).
  myaerr:ReturnValue = "Add and Remove All Msg case 66".
  RETURN ERROR myaerr.
END PROCEDURE.
Enabling stack tracing with error objects

All error objects have the ability to preserve the call stack in the \texttt{CallStack} property. (The property is populated at the time an error object is instantiated.) Examining the call stack is a valuable debugging feature, but it can potentially consume resources and it is not recommended in a production environment. ABL has an attribute on the \texttt{SESSION} handle called \texttt{ERROR-STACK-TRACE} and a startup parameter called \texttt{-errorstack} to enable or disable this feature. The default value is \texttt{FALSE} (disabled).

If \texttt{ERROR-STACK-TRACE} is \texttt{FALSE}, then the \texttt{CallStack} property of a thrown error object will be the Unknown value (?). The standard usage table for this parameter is shown below:

<table>
<thead>
<tr>
<th>Operating system and syntax</th>
<th>Windows</th>
<th>\texttt{-errorstack}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use with</td>
<td>Maximum value</td>
<td>Minimum value</td>
</tr>
<tr>
<td>Client Session</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

An error object thrown from an AppServer to a client contains the stack trace from the AppServer in its \texttt{CallStack} property. If \texttt{ERROR-STACK-TRACE} is \texttt{TRUE} on the client, the \texttt{CallStack} property will also include the client stack trace. The text "Server StackTrace:" appears at the top of the AppServer stack trace to differentiate it from the the client stack trace. In the example below, the lines in boldface represent the AppServer 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)
```
Handling Errors with CATCH Blocks

The CATCH statement defines an error handling block of code for a particular error type within a single block. This chapter describes CATCH blocks. For details, see the following topics:

• Introduction
• CATCH statement syntax
• Blocks that support CATCH blocks
• UNDO scope and relationship to a CATCH block
• Buffer scope within a CATCH block
• Flow-of-control statements in a CATCH block
• CATCH blocks within CATCH blocks
• System error message suppression with CATCH
• Handling errors from built-in ABL methods
Introduction

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

```ABL
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 (pass) any unhandled errors to the procedure (main) block, since the procedure block 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 get raised from the RUN statement. If a Progress.Lang.AppError object gets raised, the CATCH block handles the error and the error is not passed to the procedure block.

However, when this code runs, it raises a Progress.Lang.SysError because the FIND statement fails and there is no error handler present for this error type. (To handle Progress.Lang.SysError, you could add either a CATCH block for Progress.Lang.SysError or a NO-ERROR option on the FIND statement.) Since Progress.Lang.SysError is not handled, the AVM moves the error up the call stack to the procedure block. The AVM finds a compatible CATCH block on the procedure block and constructs a Progress.Lang.SysError object named eSysError, and then executes the code in the CATCH block. Error objects are deleted automatically by the AVM garbage collection mechanism.

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 error type in a block. However, a CATCH block for a specific error type will also handle error objects for its subtypes. So, it is possible that there can be more than one CATCH block to handle a particular error type in a block. If multiple CATCH blocks are compatible with the error raised, the AVM will execute the first CATCH block it encounters that is compatible with the error. For this reason, CATCH blocks should be arranged from the most specific error type to the most general error type. For example, if you had different error handling code for Progress.Lang.SysError objects and SoapFaultError objects, you would put the CATCH block for SoapFaultError objects first. Since Progress.Lang.SoapFaultError objects are a subtype of Progress.Lang.SysError objects, a CATCH block for Progress.Lang.SysError objects will also handle Progress.Lang.SoapFaultError objects.

**CATCH statement syntax**

Here is the syntax for the CATCH statement:

**Syntax**

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

**error-variable**

The variable name that references an error object derived from the built-in class Progress.Lang.ProError. Typically, you do not define the error-variable ahead of time with the DEFINE VARIABLE statement. The AVM recognizes a new variable name on the CATCH statement as a new error-variable definition. Each CATCH in an associated block must have a unique error-variable. You can reuse an error-variable name in a different associated block, as long as its type is compatible with the new definition. For example:

```
  /* This definition 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 BUTTONS OK.
  END CATCH.

  CATCH twoError AS Progress.Lang.ProError:
    MESSAGE twoError:GetMessage(1) VIEW-AS ALERT-BOX BUTTONS OK.
  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 */
CATCH oneError AS Progress.Lang.SysError:
  MESSAGE oneError:GetMessage(1) VIEW-AS ALERT-BOX BUTTONS OK.
END CATCH.

/* NOT LEGAL: Each CATCH block in an associated block must have a unique error-variable. */
CATCH oneError AS Progress.Lang.ProError:
  MESSAGE oneError:GetMessage(1) VIEW-AS ALERT-BOX BUTTONS OK.
END CATCH.
END. /* SECOND DO */

[ CLASS ] error-class

Typically Progress.Lang.SysError for system errors or Progress.Lang.AppError (or your subclass) for application errors. Optionally, you can provide the CLASS keyword.

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

```
DO TRANSACTION
  . . .
  CATCH . . . :
  . . .
END CATCH.
END.
```

The code within a CATCH block is only executed if an ERROR of type error-class (or a sub-type) is raised within the body of the associated block. This behavior is also true if any sub-routine called by the associated block returns or raises an error of type error-class. When ERROR 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. If there are variables or temp-table fields with the UNDO property (defined without the NO-UNDO option) and there is no active transaction, then these variables and fields are undone before the CATCH block executes. For more information, see the reference entries for the DEFINE VARIABLE statement and the TRANSACTION option in the DO statement in OpenEdge Development: ABL Reference.

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 BUTTONS OK.
END CATCH.
```
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 will execute the code in the first CATCH block that matches the error type. It will not execute multiple CATCH blocks. Therefore, if multiple CATCH blocks are specified, the CATCH block for 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 inherits from the same object as AppError in the class hierarchy, 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 will catch any possible error raised in ABL. */
    CATCH eError AS Progress.Lang.Error:
        /* Handler code for any error condition. */
        END CATCH.
END. /* Associated Block */
```
The compiler will issue a warning message if a block contains a **CATCH** block that is not reachable. For example, the following code will cause the compiler to issue a warning, since the **CATCH** of `myAppError` can never be executed:

```plaintext
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.
  /* Never get here, 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 */
```

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

It is valid to have both an explicit **ON ERROR** 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 error types handled by the **ON ERROR** phrase of the associated block.

### Blocks that support **CATCH** blocks

The following blocks can have a **CATCH** block:

- **DO** (if configured with **TRANSACTION** or **ON ERROR, UNDO**)
- **FOR**
- **REPEAT**
- **CATCH**
- **FINALLY**
- Procedure (.p file)
- Internal procedures
- User-defined functions
- **ON** blocks
- Constructors
- Destructors
- User-defined methods
• Property accessors

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 */
```

UNDO scope and relationship to a CATCH block

The CATCH block is an end block of its associated block. The CATCH block only executes when error is raised in the associated block and the error type specified in the CATCH statement is compatible with the type of the raised error. At the point the CATCH block executes, several steps in the error handling process are already complete, and this state affects what data is available to your CATCH block:

• Any transaction within the associated block will already be undone. In other words, changes made within the associated block to persistent data, undo variables, and undo temp-table fields have already been discarded.

• Records scoped to the associated block will be released before the CATCH block executes. (See Buffer scope within a CATCH block on page 89, for more information on buffer scope.)

• Even if no transaction is present, undoable variables and temp-table fields changed by the associated block will be restored to their last valid values before the associated block began its execution. Remember that a variable or temp-table field is undoable by default. (The NO-UNDO option on the DEFINE statement changes this default.)
The following example with step by step comments demonstrates these availability rules:

```plaintext
/* Defines an undoable variable because 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.

DO ON ERROR UNDO, LEAVE:

/* This value will be undone on ERROR! */
ASSIGN TargetCustNum = 15.

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

/* Change the database! A transaction is now active. */
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 will 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. */
IF AVAILABLE (Customer) THEN
    MESSAGE "Customer record is still available."
    VIEW-AS ALERT-BOX BUTTONS OK.
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! */

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

END. /* ELSE */

END CATCH.
END. /* DO */
```
Buffer scope within a CATCH block

In the previous section, the example showed that a record buffer scoped to the associated block of a CATCH will not be available in the CATCH block. The record buffer will be available if the ABL scoping rules have the buffer scoped to a block outside the associated block. 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.

However, note that an available buffer in a CATCH block will be affected by the UNDO operation of the associated block if that buffer is part of the transaction in the associated block. In this case, the AVM restores the buffer to the last valid state of the buffer before the associated block began its execution.

This next example demonstrates buffer availability in the common use case of looping through associated records:

```
DEFINE VARIABLE i AS INTEGER NO-UNDO.
/* An outer loop for batch updates to Customer records */
FOR EACH Customer WHERE CustNum < 2:
  /* Customer buffer scoped to FOR EACH. Will be available to inner CATCH. */
  ASSIGN Customer.Name = Customer.Name + "_changed".
  /* Update Order records for current Customer. */
  DO i = 1 TO 5 ON ERROR UNDO, LEAVE:
    IF i = 1 then
      FIND FIRST Order OF Customer.
    ELSE
      FIND NEXT Order OF Customer.
    END IF.
  END DO.
  /* Order record scoped to the DO block, which is the associated block. */
  ASSIGN Order.PromiseDate = TODAY.

  /* Nonsense code raises unique index error. DO block undone. Control passes to CATCH */
  ASSIGN Customer.CustNum = 2.
CATCH eSysError AS Progress.Lang.SysError:
  /* MESSAGE statement executes to display available Customer data. */
  Unavailable Order data raises two errors.
  Errors thrown to FOR EACH block.
  FOR EACH block undoes iteration and attempts NEXT iteration.
  Since there is not another iteration, FOR EACH completes. */
  MESSAGE Customer.Name SKIP
  Order.OrderNum Order.PromiseDate
  VIEW-AS ALERT-BOX BUTTONS OK.
END CATCH.
END. /* DO */
END. /* FOR EACH */

/* Demonstrates change made in FOR EACH block is also undone. */
MESSAGE Customer.Name VIEW-AS ALERT-BOX BUTTONS OK.
```
Here the outer loop moves through Customer records and the inner loop moves through Order records associated with the current Customer. When the inner loop fails, the Order record buffer is not available to the CATCH associated with the inner loop. The Order buffer is scoped to the inner block and after the UNDO operation completes, the record is released and unavailable. The Customer buffer is scoped outside of the associated block (the inner DO block), so the CATCH can access this buffer and display the pending change to the Customer record. Since the CATCH itself raises error by attempting to display the unavailable Order data, those errors are thrown to the FOR EACH block and the change to the Customer record is itself undone.

This next example demonstrates the need to re-find the correct record if your CATCH block needs to work with the persistent data available in the associated block at the time the AVM raised error.

The example is a variation of the example from the preceding section. This example includes an outer DO block with a reference to the Customer buffer. The re-find logic in the CATCH block is now in the IF AVAILABLE(Customer) branch of the IF statement.

The following is the code:

```/* Defines an undoable variable because 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.

/* Outer DO */
DO ON ERROR UNDO, LEAVE:
    /* Customer buffer is scoped outside of the associated block */
    FIND Customer WHERE CustNum = TargetCustNum.
    DISPLAY Customer.Name WITH SIDE-LABELS.

    /* Inner DO and also the associated block*/
    DO ON ERROR UNDO, LEAVE:
        /* This value will be undone on ERROR! */
        ASSIGN TargetCustNum = 15.
        /* Find a Customer */
        FIND Customer WHERE Customer.CustNum = TargetCustNum.

        /* Change the database! A transaction is now active. */
        ASSIGN Customer.Name = Customer.NAME + " And Much More".

        /* Confirm change to persistent field. */
        MESSAGE "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 will not execute. */
        DISPLAY Customer.CustNum SKIP Customer.Name SKIP OrderNum SKIP OrderStatus VIEW-AS TEXT WITH SIDE-LABELS.

        CATCH eSysError AS Progress.Lang.SysError:
            /* Confirm if Customer record is available in CATCH. */
            IF AVAILABLE (Customer) THEN DO:
                MESSAGE "Customer record is still available."
                VIEW-AS ALERT-BOX BUTTONS OK.
            /* Re-Find the Customer to ensure buffer scoped to outer DO */
```
Flow-of-control statements in a CATCH block

The code in any CATCH block can contain an explicit flow-of-control directive, including LEAVE, NEXT, RETRY, RETURN, or THROW. If you use RETRY or THROW, then the UNDO option is required. Since CATCH is an undoable block, LEAVE, NEXT, and RETRY without a label applies to the CATCH block itself and not the associated block.

If you want LEAVE, NEXT, or RETRY to apply to the associated block of a CATCH block, you must use the label syntax for these statements.

An explicit UNDO, THROW in a CATCH block causes the AVM to raise ERROR in the block that encloses the associated block of the CATCH block; not the associated block itself.

In this example, LEAVE in the CATCH applies to the CATCH:

```
DEFINE VARIABLE iOrdNum AS INTEGER.
DEFINE VARIABLE lSomeCondition AS LOGICAL.

FOR EACH Customer:
  UPDATE iOrdNum.
  FIND Order WHERE Order.CustNum = Customer.CustNum
      AND Order.OrderNum = iOrdNum. /* Can fail and raise ERROR */
  DISPLAY Order.OrderNum Order.ShipDate. /* don't get here if FIND fails */
CATCH eSysError AS Progress.Lang.SysError:
  MESSAGE "Order " iOrdNum " does not exist for Customer ".
  /* This LEAVE applies to the CATCH.
  Execution will retry the same customer */
  IF lSomeCondition THEN DO:
    UNDO, LEAVE.
  END.
  ...
  /* more statements in the CATCH that will execute if UNDO, LEAVE
didn't execute */
END CATCH.
END. /* FOR EACH Customer */
```
In this example, the procedure gives the user three chances to get the right order number:

```
DEFINE VARIABLE iOrdNum as INTEGER.
DEFINE VARIABLE iTries as INTEGER.
ASSIGN iTries = 1.
/* Associated block has a label */
blk1:
FOR EACH Customer:
    /* User input statement means default error handling is UNDO, RETRY */
    DISPLAY Customer.Name.
    UPDATE iOrdNum.
    /* Will fail and raise ERROR if user enters invalid Order number */
    FIND Order WHERE Order.CustNum = Customer.CustNum
    AND Order.OrderNum = iOrdNum.
    ...
CATCH eSysError AS Progress.Lang.SysError:
    MESSAGE "Order " iOrdNum " does not exist for this Customer.".
    /* Note that iTries is not reset on UNDO of the FOR EACH because it
    is not referenced in the FOR EACH block. */
    IF iTries <= 3 THEN DO:
        ASSIGN iTries = iTries + 1.
    END.
    ELSE DO:
        MESSAGE "Too many tries for this Customer".
        ASSIGN iTries = 1.
        /* Leave the CATCH. Execution will resume with the NEXT
        iteration of the FOR EACH */
        UNDO, NEXT blk1.
    END.
END CATCH.
END. /* blk1 FOR EACH */
```

Because this example includes input from the user, the default error handling is RETRY for the FOR EACH block. The CATCH block overrides the RETRY option only if the user has failed three times to enter a correct Order number for a particular Customer. Then, the UNDO, LEAVE blk1 statement does not UNDO the associated block because that step has already occurred. The NEXT directive executes and the FOR EACH resumes execution with the next iteration of the block.

**Note:** Note that when you are working with nested blocks, there is a relationship between the scope of a LEAVE (or NEXT) directive and the scope of the UNDO. You cannot LEAVE or NEXT to a context outside the scope of the UNDO while the potential for the UNDO is in effect. If the UNDO action has already occurred, you can use LEAVE and NEXT to go to a higher context.

If there is no explicit flow-of-control statement in the CATCH block, the AVM will leave the CATCH block and execute the default error action for the associated block after executing the last statement in the CATCH block and any code within a FINALLY block. This means RETRY for all blocks, when no I/O blocking (user input) statements are present, the AVM prevents infinite looping by changing the RETRY to NEXT for iterating blocks or LEAVE for non-iterating blocks.
In the following code, if an `Acme.Error.myAppError` is caught the explicit `UNDO, THROW` statement causes the caught error to be thrown to the block enclosing the `FOR EACH` (remember that `UNDO, THROW` in a `CATCH` means leave the associated block, then `THROW`). However, if a `Progress.Lang.SysError` is caught the AVM will execute a `NEXT` on the `FOR EACH` block, as shown:

```plaintext
FOR EACH Customer:
  /* FOR EACH code */
  CATCH eSysError AS Progress.Lang.SysError:
    /* Handler code for SysError condition */
    /* RETRY on FOR EACH after leaving the CATCH, which becomes NEXT if there are no I/O statements.*/
  END CATCH.

CATCH myAppErr AS Acme.Error.myAppError:
  /* Handler code for myAppError condition */
  UNDO, THROW myAppErr. /* THROW error to block enclosing FOR EACH */
END CATCH.
END.
```

### CATCH blocks within CATCH blocks

The `CATCH` block is an undoable block with implicit `ON ERROR UNDO, THROW` error handling. You cannot explicitly override the `ON ERROR` directive for a `CATCH` block. If a statement within the `CATCH` block raises `ERROR` and there is no nested `CATCH` block, the `CATCH` block will be undone, and the `ERROR` will be raised in the block that encloses the associated block of the `CATCH`. So a statement that raises `ERROR` within a `CATCH` block causes the following to occur:

1. **UNDO** the `CATCH` block.
2. **Leave** the `CATCH` and its associated block.
3. **THROW** the error to the enclosing block. If the `CATCH` block is at the routine level, then the error gets thrown to the caller of the routine. The same behavior occurs for an explicit `UNDO, THROW` statement in a `CATCH` block.
A `CATCH` block can have a `CATCH` block within it. In this case, the contained `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:

```plaintext
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 in the `DO` block. The top-level `CATCH` block associated with the `DO` block handles application errors. The nested `CATCH` block handles any system errors that occur in the top-level `CATCH` block. If `ERROR` is raised, then the `UNDO, THROW` statement in the nested `CATCH` block immediately passes control to the block enclosing the associated block of the top-most `CATCH` block in a set of nested `CATCH` blocks. If 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.

If there is a `FINALLY` block in the associated block, the `FINALLY` code will be executed before `ERROR` gets raised in the block enclosing the associated block.

### System error message suppression with CATCH

The presence of a `CATCH` in an undoable block causes the AVM to suppress system error messages for all statements within the block, in the same way that the `NO-ERROR` option works on individual statements. If there is a `CATCH` on `Progress.Lang.SysError`, the messages will be added to the `Progress.Lang.SysError` object that is available in the `CATCH`. If there is no `CATCH` on `Progress.Lang.SysError`, and the `Progress.Lang.SysError` is not re-thrown (by way of `ON ERROR UNDO, THROW` on the block), the error messages from the `Progress.Lang.SysError` object will be written to the current output destination and the AVM will execute the `ON ERROR` phrase for the block.
In this example, a CATCH handles the error and the default error message is suppressed:

```abl
DEFINE VARIABLE myInt as INTEGER INITIAL 5.
DO ON ERROR UNDO, LEAVE:
  FIND Customer WHERE CustNum=1000. /* raises ERROR and throws Progress.Lang.SysError
  Error message suppressed and execution goes to CATCH */
  MESSAGE "After Find". /* won't get here */
CATCH eSysError AS Progress.Lang.SysError:
  MESSAGE "Inside CATCH block" VIEW-AS ALERT-BOX.
  /* Leave the CATCH, then the DO */
END CATCH.
END.
```

In this example, there is no CATCH block that handles the error and the error message is not suppressed:

```abl
DEFINE VARIABLE myInt as INTEGER INITIAL 5.
DO ON ERROR UNDO, LEAVE:
  FIND Customer WHERE CustNum=1000. /* raises ERROR and displays "Customer record not on file. (138)"
  UNDO, LEAVE the block */
  MESSAGE "After Find". /* won't get here */
CATCH ae AS Progress.Lang.AppError:
  MESSAGE ae:GetMessage(1) VIEW-AS ALERT-BOX.
END CATCH.
END.
```

Handling errors from built-in ABL methods

Traditional error handling treats errors arising from the methods on built-in system handles as warnings. The **ERROR** condition is not raised, but you can use the **NO-ERROR** option to trap the error messages in the **ERROR-STATUS** system handle.

When any CATCH block is present in an associated block, built-in ABL methods raise error. If no CATCH block is present in the associated block that handles ABL system errors, the error is still raised and handled by the implicit or explicit **ON ERROR** phrase of the associated block.
For example, consider this code:

```plaintext
DO ON ERROR UNDO, THROW:

    DEFINE VARIABLE hSocket AS HANDLE NO-UNDO.
    CREATE SOCKET hSocket.
    hSocket:CONNECT ("-H localhost -S 3333") /* NO-ERROR */.

    CATCH eSysError AS Progress.Lang.SysError:
        DISPLAY "CATCH handles connection error on hSocket." SKIP
        "Messages..." SKIP
        eSysError:GetMessage(1) FORMAT "x(70)" SKIP
        eSysError:GetMessage(2) FORMAT "x(70)".
    END CATCH.

    END. /* DO */

    IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
        DISPLAY "ERROR-STATUS handle traps connection error on hSocket." SKIP
        "Message..." SKIP
        ERROR-STATUS:GET-MESSAGE(1) FORMAT "x(70)" SKIP
        ERROR-STATUS:GET-MESSAGE(2) FORMAT "x(70)".
    END. /* DO */
```

When the `CONNECT` method fails, the `CATCH` executes and displays two messages. The `ERROR-STATUS` handle is not used, so the second `DISPLAY` statement does not execute.

Remove the comments from the `NO-ERROR` option on the `CONNECT( )` method and run the code again. This time the `CATCH` block does not execute. The `ERROR-STATUS` system handle traps the error messages, so the second `DISPLAY` statement executes.
Raising errors with THROW

This manual has shown many examples of the THROW directive along with all of the ABL language syntax that supports THROW. This chapter adds to the discussion by focusing on some additional use cases for THROW.

For details, see the following topics:

- Propagating errors up the stack
- THROW with error objects
- THROW as a default for blocks
- THROW with user-defined functions

Propagating errors up the stack

The best practice for most error handling is to handle the error locally. For both traditional and structured error handling, ABL default behavior supports handling the error as close to where the error occurred as possible. ABL attempts to minimize the amount of work that has to be undone, and it provides branching options to help resume execution.

In practice, localized error handling means that each block should have enough CATCH blocks to handle any error that can be reasonably expected to occur within the block. This makes code more readable and maintenance straightforward.
However, there are also numerous use cases where handling errors in a central location makes more sense. For example, suppose you have a code module with many blocks that can fail with the same type of error. If there is no advantage to a local `CATCH` block, and your error handling code is the same for all blocks in the module, then `THROW` all the errors up the call stack to a central location where a single `CATCH` can handle them all.

In many cases, you can provide local `CATCH` blocks that require specific error-handling behavior to support your business logic. But even if your application handles all predictable errors, this still leaves unhandled system errors as a potential problem. You can use `THROW` directives to move unhandled system errors from the block in which they occur to outer blocks, until they reach some central point where you have a common `CATCH` block set to handle them—perhaps, for example, by providing common logging and graceful exit behavior.

The technique of directing a containing context to handle errors raised in an inner context is often described as *propagating errors up the call stack*. The remaining sections in this chapter examine the use of `THROW` to apply this technique in your application code.

## THROW with error objects

As part of the `ON ERROR` phrase, `UNDO`, `THROW` causes the AVM to undo the current transaction, suppress the display of system error messages, create and populate an error object, and throw that error object. Assuming that the errors are not handled within the current block, `THROW` directs the block enclosing the current block to handle the errors. In other words, it propagates the errors up the call stack. For example:

```plaintext
DO TRANSACTION ON ERROR UNDO, THROW:
   FIND FIRST Customer WHERE Customer.CustNum = 1000.
   /* Fails and raises error. */
END.
```

The `THROW` directive on the `UNDO` statement allows you to raise specific application errors in the block in which the statement occurs. For example:

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

In the preceding example, a `Progress.Lang.AppError` is created with parameters that add a message and a message number to the new object.
THROW compared with RETURN ERROR

By way of contrast, the RETURN ERROR statement can also be used to raise an AppError object. It raises ERROR and also provides the ability to specify a CHARACTER string that can be accessed in the caller (usually the ReturnValue property of the error object or the RETURN-VALUE function). The RETURN ERROR statement allows you to return an error object. For example:

```
IF CurrentTime > ClosingTime THEN
    RETURN ERROR NEW Progress.Lang.AppError("Can't take a delivery order after closing time.").
```

RETURN ERROR does not provide the UNDO action as does the UNDO, THROW statement.

The following language elements support a RETURN ERROR error-object-expression action:

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

THROW as a default for blocks

For those who want to apply the THROW directive broadly for a file (procedure or class) or for an entire application, OpenEdge provides options that make UNDO, THROW the error-handling behavior for blocks where no explicit directive is specified and default behavior (UNDO, RETRY or UNDO, LEAVE, depending on the block type) would otherwise be in effect. These options are:

- BLOCK-LEVEL ON ERROR UNDO, THROW statement
- ROUTINE-LEVEL ON ERROR UNDO, THROW statement
- -undothrow n startup parameter

For troubleshooting and other purposes, it is important to be able to determine whether a given piece of compiled code uses one of these directives. To support this requirement, the AVM records usage of the block-level and routine-level error-handling statements in:

- An attribute of the RCODE-INFO system handle
- XREF and XREF-XML output

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. 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
• Property accessor
• REPEAT
• FOR
• DO TRANSACTION

The following blocks are not affected:
• Any block for which an error-handling directive is explicitly specified
• Simple DO 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:

• Include the BLOCK-LEVEL ON ERROR UNDO, THROW statement in all your procedure and class files.
• Add a CATCH block for the Progress.Lang.Error interface to your startup procedure block.
• For each basic block, decide whether or not an explicit THROW is appropriate for that block.
This example is simple but illustrates the design pattern:

```
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 */
MESSAGE "Application completed execution successfully."
    VIEW-AS ALERT-BOX BUTTONS OK.
CATCH eAnyError AS Progress.Lang.Error:
    MESSAGE "Unexpected error occurred..."
    "Logging information..."
    "Exiting application..."
    VIEW-AS ALERT-BOX BUTTONS OK.
    QUIT.
END CATCH.
```

**ROUTEINE-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: the main blocks of ABL routines. 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
- Property accessor

The following blocks are **not** affected:

- Any block for which an error-handling directive is explicitly specified
- **REPEAT**
• FOR
• DO TRANSACTION
• Simple DO 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 n 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 this 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 the value 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 101.
- -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, the parameter supersedes them. See BLOCK-LEVEL ON ERROR UNDO, THROW statement on page 99.

Using this parameter with an argument other than 1 or 2 results in an error and immediate termination of the operation. The parameter is ignored if it is used in a context other than that of a COMPILE statement or an on-the-fly compilation.
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. This attribute is of type CHARACTER and has the following possible values:

- "ROUTINE-LEVEL"
- "BLOCK-LEVEL"
- Null (""), if neither directive was 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>xX6q2Fj+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.
THROW with user-defined functions

The user-defined function, which is defined by the FUNCTION statement, returns a value of a specific data type as its primary function. The RETURN statement is the statement you use to specify in the function body what value to return to the caller. This fact makes it impossible to use the RETURN ERROR statement in the same way as it is used in other blocks.

RETURN ERROR in a user-defined function does not raise error in the caller. Instead, it sets the target variable of the function to the unknown value. Therefore, you could perform error checking on a function call by checking for the unknown value after a function call. This technique only works if the function uses the RETURN ERROR statement and the target variable has a value other than the unknown value at the time of the function call. For example:

```
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."
```

Structured error handling provides more ways to raise error from user-defined functions.

Throwing system errors

With structured error handling, you can raise error in the caller by throwing an error from a CATCH block on the main function block. A CATCH block in the caller can then handle system error, as shown:

```
/* First technique for returning a system error from a user-defined function. */
DEFINE VARIABLE FuncReturn AS LOGICAL NO-UNDO.

FUNCTION ReturnSysError RETURNS LOGICAL:
  FIND FIRST Customer WHERE CustNum = 1000.
  CATCH mySysError AS Progress.Lang.SysError:
    UNDO, THROW mySysError.
  END CATCH.
END FUNCTION.

ASSIGN FuncReturn = ReturnSysError().
CATCH mySysError AS Progress.Lang.SysError:
  DISPLAY "Error message returned from function: "
  mySysError:GetMessage(1) FORMAT "X(60)" SKIP.
END CATCH.
```
You might want to upgrade your functions to use this technique without upgrading the traditional error handling used in the caller. You can use the NO-ERROR option on the ASSIGN statement of the target variable. The AVM automatically moves the information found in the SysError object to the ERROR-STATUS system handle. For example:

```asi
/* Second technique for returning a system error from a user-defined function. */

DEFINE VARIABLE FuncReturn AS LOGICAL NO-UNDO.

FUNCTION ReturnSysError RETURNS LOGICAL:
    FIND FIRST Customer WHERE CustNum = 1000.
    CATCH mySysError AS Progress.Lang.SysError:
        UNDO, THROW mySysError.
    END CATCH.
END FUNCTION.

ASSIGN FuncReturn = ReturnSysError() NO-ERROR.

IF ERROR-STATUS:ERROR THEN
    DISPLAY "Error message returned from function: "
    ERROR-STATUS:Get-Message(1) FORMAT "X(60)".
END IF.
```

The ROUTINE-LEVEL ON ERROR UNDO, THROW statement can also be used to change the default error handling for all user-defined functions in a procedure or class file. In the following example, notice that when the FIND fails the presence of the implicit THROW suppresses display of the error message at the statement level:

```asi
/* Third technique for returning a system error from a user-defined function. */

ROUTINE-LEVEL ON ERROR UNDO, THROW.

DEFINE VARIABLE FuncReturn AS LOGICAL NO-UNDO.

FUNCTION ReturnSysError RETURNS LOGICAL:
    FIND FIRST Customer WHERE CustNum = 1000.
END FUNCTION.

ASSIGN FuncReturn = ReturnSysError().

CATCH mySysError AS Progress.Lang.SysError:
    DISPLAY "Error message returned from function: "
    mySysError:GetMessage(1) FORMAT "X(60)" SKIP.
END CATCH.
```
Throwing application errors

Throwing application errors is a little different. In the first example, you provide a constructor on the UNDO statement to instantiate the AppError. In the caller, you retrieve the error message from the ReturnValue property of the AppError object, as shown:

```plaintext
/* First technique for returning an application error from a user-defined function. */
DEFINE VARIABLE FuncReturn AS LOGICAL NO-UNDO.
FUNCTION ReturnAppError RETURNS LOGICAL:
    FIND FIRST Customer WHERE CustNum = 1000.
    RETURN TRUE.
    CATCH anyError AS Progress.Lang.ProError:
        UNDO, THROW
    END CATCH.
END FUNCTION.
ASSIGN FuncReturn = ReturnAppError().
CATCH myAppError AS Progress.Lang.AppError:
    DISPLAY "Error message returned from function: " myAppError:ReturnValue FORMAT "X(60)".
END CATCH.
```

In the second example, to return the AppError to a caller using traditional error handling, again you use the NO-ERROR option on the caller. In an AppError object exists and the ReturnValue property is populated, the AVM always makes that value available in the RETURN-VALUE function, as shown:

```plaintext
/* Second technique for returning an application error from a user-defined function. */
DEFINE VARIABLE FuncReturn AS LOGICAL NO-UNDO.
FUNCTION ReturnAppError RETURNS LOGICAL:
    FIND FIRST Customer WHERE CustNum = 1000.
    RETURN TRUE.
    CATCH anyError AS Progress.Lang.ProError:
        UNDO, THROW
    END CATCH.
END FUNCTION.
ASSIGN FuncReturn = ReturnAppError() NO-ERROR.
IF ERROR-STATUS:ERROR THEN
    DISPLAY "Error message returned from function: " RETURN-VALUE FORMAT "X(60)".
```
Using FINALLY End Blocks

FINALLY end blocks provide end of block processing for basic blocks. Processing occurs on each iteration of a block, even if that iteration resulted in error. For details, see the following topics:

- Introduction
- Syntax
- Behavior
- Examples

Introduction

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

The `FINALLY` block executes after:

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

The **FINALLY** block will not execute if:

• **STOP** condition is raised and not handled
• **QUIT** statement is in effect and it is not handled

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

The purpose of a **FINALLY** block is to hold clean-up code that must execute regardless of what else executed in the associated block. It can include code to delete dynamic objects, write to logs, close outputs, and other routine tasks. Because it executes even if the **ERROR** condition is raised, the **FINALLY** block is also a useful part of a structured error handling scheme.

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

**Syntax**

Here is the syntax for a **FINALLY** block:

**Syntax**

```
block-statement   FINALLY:
    .
    .
    .
    END  [ FINALLY ].
block-end-statement.
```

**block-statement**

One of the following ABL blocks:

• **DO** block
• **FOR** block
• **REPEAT** block

**block-end-statement**

The **END** statement terminating the enclosing associated block of the **FINALLY** block.
Behavior

The topics in this section describe the key properties of the FINALLY block:

- Execution of a FINALLY block on page 109
- Transaction scope in a FINALLY block on page 109
- Flow-of-control in a FINALLY block on page 110
- STOP or QUIT conditions with FINALLY blocks on page 110

Execution of a FINALLY block

The FINALLY block executes as an end block of the associated block. The FINALLY block executes once for each iteration of its associated block. This means you can only execute the FINALLY block during the execution of the associated block. The FINALLY block executes whether the associated block executes successfully or raises ERROR.

Transaction scope in a FINALLY block

The transaction of the associated block is either complete (success) or undone (failure) when FINALLY executes. Therefore, any UNDO statement within the FINALLY block will only undo the work in the FINALLY block.

The FINALLY block is an undoable block with implicit ON ERROR UNDO, THROW error handling. You cannot explicitly override the ON ERROR directive for a FINALLY block. If a statement within the FINALLY block raises ERROR, the FINALLY block will be undone, and ERROR will be raised in the block that encloses the associated block of the FINALLY block. Error is not raised in the associated block. Otherwise, infinite looping could occur.

A statement that raises ERROR within a FINALLY end block causes the following to occur:

1. UNDO the FINALLY block
2. LEAVE the associated block
3. THROW the error to the block enclosing the associated block

The same behavior occurs for an explicit THROW statement in a FINALLY block.

Buffer scope in a FINALLY block

Buffers scoped to the associated block of the FINALLY block will not be available when the FINALLY block. 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.
Flow-of-control in a FINALLY block

The code in any FINALLY block can contain explicit flow-of-control options: LEAVE, NEXT, RETRY, RETURN, or THROW. Since FINALLY is an undoable block, LEAVE, NEXT, and RETRY without a label apply to the FINALLY block itself and not to the associated block.

If you want LEAVE, NEXT, or RETRY to apply to the associated block, use label syntax with these statements. Flow of control statements in a FINALLY block override pending flow of control from a CATCH block.

STOP or QUIT conditions with FINALLY blocks

If the AVM detects a STOP or QUIT condition in the associated block, the FINALLY block will not run and the AVM processes the condition. If the associated block has an ON STOP or ON QUIT phrase, then the STOP or QUIT condition is handled and released by the time the AVM is ready to execute the FINALLY block, and the FINALLY block is executed. See the ON STOP and ON QUIT reference entries in OpenEdge Development: ABL Reference for a description of the STOP and the QUIT condition behavior and handling.

Examples

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 (LEAVE, NEXT, RETRY, RETURN, or THROW) options 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
   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.
   /* 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:
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:
Example 3

In Example 3, after ERROR is raised, execution goes to the CATCH block, which re-throws the error. However, the FINALLY block executes before the error goes to the CATCH block associated with the procedure block:

```plaintext
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 FINALLY block." VIEW-AS ALERT-BOX BUTTONS OK.
   END FINALLY.

END. /* DO */

CATCH eSysError AS Progress.Lang.SysError:
   MESSAGE "Out of DO block and inside CATCH block for procedure block"
   VIEW-AS ALERT-BOX BUTTONS OK.
END CATCH.
```

If you run this code, you see the following messages:
Out of DO block and inside CATCH block for procedure block.
Index

A
Add additional index entries here (if needed) 71
   Nest index terms for sub entries 71
AddMessage method 75
AppError constructors 76
Application errors 25
Application errors, defined 19
Associated block types 86

B
Basic blocks 30
Block labels 43
BLOCK-LEVEL ON ERROR UNDO, THROW statement 99
Blocks 26, 29–32, 35
   basic 30
   CATCH 30
   class 31
   constructor 31
   database trigger 31
   database trigger procedure 31
   destructor 31
   DO 30
   end 30
   FINALLY 30
   FOR 30
   internal procedure 31
   labels 35
   Method 31
   ON 31
   Procedure 31
   property accessor 31
   REPEAT 30
   routine-level 30
   summary 32
   types 29
   undoable 26
   user-defined function 31
Branching defaults 34
Branching options 34
Built-in method warnings 45

C
CallStack 79
CallStack property 70
CATCH block 30, 82–83, 86–87, 89, 91, 93–95
   and built-in methods 95
   and FINALLY block 93
   associated block types 86
   buffer scope 89
   compiler warnings 83
   error-class 83
   error-variable 83
   flow-of-control statements 91
   introduced 82
   nested CATCH blocks 93
   rules 82
   syntax 83
   system error suppression 94
   THROW directive 82
   transaction scope 87
   UNDO scope 87
CATCH statement 27
Class block 31
Classes 72–74
   Progress.Lang.AppError 74
   Progress.Lang.ProError 72
   Progress.Lang.SoapFaultError 73
   Progress.Lang.SysError 72
Condition handling, defined 19
Condition, defined 19
Conditions 19, 21–22, 51–52, 58
   ENDKEY 22
   QUIT 22, 58
   STOP 19, 22, 51–52
   database disconnection 52
Connections, lost 52
Constructor block 31

D
Database 30, 52
   , lost connections 52
   triggers 30
Default error handling 19, 23
   defined 19
   traditional 23
Destructor block 31
DO block 30

OpenEdge Development: Error Handling 115
E

End blocks 30
END statement 108
END-ERROR key 22
ENDKEY condition 22
ENDKEY key 22
ERROR attribute 45
Error classes 69
ERROR condition, described RETURN ERROR statement 22
ERROR condition, raising 48
Error handling 19–20
defined 19
models 20
ERROR key 22
Error messages 24, 39, 63
suppressing 24
Error number 39
Error objects, defined 25
ERROR option 40, 48, 53, 59
ON ERROR phrase 40
ON QUIT phrase 59
ON STOP phrase 53
RETURN statement 48
Error string 39
Error-handling models 63, 103
compared 63
determining usage in r-code 103
ERROR-OBJECT-DETAIL 45, 73
attribute 45
property 73
ERROR-STACK-TRACE 79
ERROR-STATUS 24, 45, 73
system handle 24, 45
Errors 19, 38, 44, 69, 97
defined 19
propagating 97
statement-level 38
suppressing 38
throwing 97
types 69
Exception, defined System errors, defined 19

F

FINALLY (continued)
statement 25, 65
FINALLY block 107–110
and flow-of-control statements 110
and STOP condition 110
behavior 109
block-statement option 108
buffer scope 109
execution rules 107
QUIT condition 110
syntax 108
transaction scope 109
FINALLY statement 108
block-end-statement 108
FOR block 30
Functions 25
RETURN-VALUE 25
Functions, user-defined 49
Functions, user-defined and THROW 104

G

GET-MESSAGE method 45
GET-NUMBER method 45
GetMessage method 70
GetMessageNum method 70

I

Interfaces 70
Progress.Lang.Error 70
Internal procedure block 31

K

Keys 22, 51
END-ERROR 22
ENDKEY 22
ERROR 22
STOP 22, 51

L

LEAVE option 34, 40, 53, 59
ON ERROR phrase 40
ON QUIT phrase 59
ON STOP phrase 53
UNDO statement 34
Lost database connections 52
M

Messages 39
during execution 39
Method block 31

N

Nested CATCH blocks 93
NEXT option 34, 40, 53, 59
ON ERROR phrase 40
ON QUIT phrase 59
ON STOP phrase 53
UNDO statement 34
NO-APPLY option 40, 48, 53, 59
ON ERROR phrase 40
ON QUIT phrase 59
ON STOP phrase 53
RETURN statement 48
NO-ERROR option 39, 44, 63
NO-UNDO option 23, 33
NUM-MESSAGES attribute 45
NumMessages property 70

O

ON block 30
ON ENDKEY phrase 99
ON ERROR phrase 24, 34, 40, 99
default branching 34
error-object-expression 40
ON QUIT phrase 59, 99
error-object-expression 59
ON STOP phrase 52–53, 99
database disconnection 52
error-object-expression 53
ONERROR phrase 98
THROW option 98
OpenEdge messages 39

P

Phrases 24, 40
ON ERROR 24, 40
Procedure block 31
Progress messages file 39
Progress.Lang.AppError class 74
Progress.Lang.AppError object 82
Progress.Lang.Error interface 70
Progress.Lang.ProError class 72
Progress.Lang.SoapFaultError class 73
Progress.Lang.SysError class 72
promsgs file 39
Propagating errors 97
Property accessor block 31

Q

QUIT condition 22, 58–59, 110
default handling 58
overriding default handling 59
QUIT statement 22

R

r-code 103
determining error-handling characteristics of 103
Raising ERROR 63
Remote procedures 48
RETURN statement 48
RemoveMessage method 48
REPEAT block 30
RETRY option 34, 40, 53, 59
ON ERROR phrase 40
ON QUIT phrase 59
ON STOP phrase 53
UNDO statement 34
RETURN ERROR 99
compared with THROW 99
RETURN ERROR and user-defined functions 49
RETURN ERROR statement 25, 63, 98
and AppError objects 98
RETURN option 34, 40, 53, 59
ON ERROR phrase 40
ON QUIT phrase 59
ON STOP phrase 53
Return value 48
RETURN statement 48
RETURN-VALUE function 25
ReturnValue property 75
Routine-level blocks 30, 65, 101
default error-handling for 65, 101
ROUTINE-LEVEL ON ERROR UNDO, THROW statement 65, 101
S
SESSION handle 79
Severity property 70, 75
SOAP faults 37, 73
  Progress.Lang.SoapFaultError class 73
Soap-Fault-Detail 73
SoapFault property 73
Stack tracing 79
Startup parameter undothrow to apply structured error handling at compile time 65
Startup procedure 51
  retrying 51
Statement errors 38
Statements 22, 25, 27, 35, 47
  CATCH 27
  FINALLY 25
  QUIT 22
  RETURN ERROR 22, 25
  STOP 22
  UNDO 35, 47
STOP condition 19, 22, 51–53, 110
database disconnection 52
default handling 51
  overriding default handling 53
STOP condition, timed 55
STOP key 22, 51
STOP statement 22
STOP-AFTER phrase 55
Structured error handling 20, 25
defined 20
  overview 25
Suppressing error messages 24
Suppressing errors 44
System handles 24
  ERROR-STATUS 24

T
THROW and user-defined functions 104
THROW directive 97, 99
  compared with RETURN ERROR 99
THROW option 34, 40, 63
  ON ERROR phrase 40
Throwing errors 65, 71, 97
  from AppServer to ABL client 65, 71
Timed STOP condition 55
Traditional error handling 20, 23, 25, 37
Application errors 25
default error handling 23
defined Structured error handling 20
defined 20
  overview 23
TRANSACTION option 23, 33
Transaction, defined 33
Transactions, undoing 33
Try and catch 26
Try and catch, defined 20

U
UNDO action 23
UNDO concept 33
UNDO option 59
  ON QUIT phrase 59
UNDO processing 51–52, 58
database disconnection 52
UNDO statement 35, 47, 63, 98–99
THROW option 98
UNDO-THROW-SCOPE attribute 103
Undoable blocks 26
Undoable temp-table fields 33
Undoable variables 33
Undoing transactions 33
Undoing work 47
undothrow startup parameter 65, 102
User-defined function block 31
User-defined functions 49
User-defined functions and THROW 104
User-interface trigger block 31

W
Warnings for built-in methods 45
Web service call 73