Progress®



Basic ABL

OpenEdge®

Copyright

© 2021 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.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

Learn about ABL....9

What is ABL?....9
Run example code....10
Connect to a database....10

Concepts and Syntax....13

ABL basic syntax....13
Procedure file....14
Classes and interfaces....14
Statements....15
Blocks....15
Comments....17
Variables and data types....17
Compound assignment operators....20
Specify display format for variables....21
Data type processing....22
Control flow....44
IF ... THEN ... ELSE statement....44
CASE statement....44
DO statement....45
FOR statement....46
REPEAT statement....47
Expressions and operators....48
Built-in functions....48
Handle-based and class-based objects....49
Code for portability....50

Procedures and User-defined Functions....51

ABL procedures....51
User-defined functions....55

Compile and Run....57

R-code and the AVM....57
COMPILE statement....58
Set compiler options....58
RUN statement....63
Run using the command line....63

Work with the OpenEdge Database.....65
Tables and buffers.....65
Data access.....66
Queries.....68
Transactions.....70
Record locks.....72
Update records.....73
Online database changes using ABL.....74

Temp-tables and Datasets.....75
Temp-tables.....75
Include files for temp-table definitions.....77
Empty a temp-table.....77
Copy records from a database to a temp-table.....78
ProDataSets.....78
ProDataSet concepts.....79
Define and use a dataset.....80

Preface

Purpose

This guide contains information for the Basic ABL track, a path for learning ABL (Advanced Business Language).

Audience

This guide is intended for new ABL developers.

Organization

This guide is organized into the following sections:

- Learn about ABL
- Concepts and syntax
- Procedures and user-defined functions
- Compile and run
- Work with the OpenEdge database
- Temp-tables and datasets

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

Purpose

This guide contains information for the Basic ABL track, a path for learning ABL (Advanced Business Language).

Audience

This guide is intended for new ABL developers.

Organization

This guide is organized into the following sections:

- Learn about ABL
- Concepts and syntax
- Procedures and user-defined functions
- Compile and run
- Work with the OpenEdge database
- Temp-tables and datasets

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

1

Learn about ABL

Advanced Business Language (ABL) is the development language used in OpenEdge. The following topics introduce you to ABL and provide information on how to start learning ABL as quickly as possible.

For details, see the following topics:

- What is ABL?
- Run example code
- Connect to a database

What is ABL?

Advanced Business Language (ABL) is the development language used in OpenEdge. The language, typically classified as a fourth-generation programming language, uses an English-like syntax to simplify software development especially in the area of databases and business logic. Unlike other languages, ABL empowers developers to manage relational data in a way that best matches their business needs, significantly boosting productivity. An OpenEdge developer can create APIs for .NET, JavaScript, Java clients, Java messaging and even map database schema to XML formats, while still leveraging the efficiency of ABL.

ABL supports two programming models, procedural and object-oriented. Both models can be used independently or together.

ABL code is compiled into machine independent byte code known as r-code. This byte code can run on multiple operating systems using the ABL Virtual Machine (AVM).

This track focuses on the basics of ABL and is applicable to both procedural and object-oriented programming. The track assumes that you already have knowledge and experience of programming in another high-level language. It introduces the basics of ABL and provides references to other documentation where you can find more information.

For more information, see Introducing ABL.

Run example code

Code examples are provided throughout this track to help you learn ABL. You can try the examples in the ABL Dojo or you can set up a development environment and try them out there.

The ABL Dojo allows you to quickly try out code without having to install any software. All you need is a browser to start writing and executing your code. You access the ABL Dojo with the following URL: https://abldojo.services.progress.com/.

For more information on the ABL Dojo see https://www.progress.com/openedge/features/abl-dojo.

You can also set up your own development environment and use the sample Sports2020 database, or create your own database to work with. More information on setting up your development environment can be found in the Progress OpenEdge documentation:

- *Install OpenEdge*
- *Progress Developer Studio for OpenEdge*
- *Manage the OpenEdge Database*

The *Develop ABL Applications* and *ABL Reference* books can also be helpful when working with ABL.

Connect to a database

The power of ABL is in the simplicity of working with databases. During application development and when running your application, it must be connected to a database. The ABL Dojo is already connected to a sample database so there are no additional steps you need to take when trying your code in the Dojo.

If you are setting up your own development environment however, you need a sample database to work with. If your application database is not yet available, there are sample databases that you can copy and use. For more information, see "Create a database with the PRODB utility" in *Manage the OpenEdge Database*.

Before connecting an application to a database you may have to start the database server. For more information on starting a database server, see "Start up and shutdown a database server" in *Manage the OpenEdge Database*. You can also connect to a database without starting a database server by running in single-user mode. In this mode only one user can access the database.

There are two ways to connect the database to the application:

- CONNECT statement — The CONNECT statement is a simple, programmatic mechanism to establish a connection to one or more databases from within an ABL procedure or class. This step must be done in a file before any files can use the database.

  The following code connects to a sample database in the OpenEdge installation directory in single-user mode:

  ```
  CONNECT WRK/db/Sports2020 -1.
  ```

  The following code connects to a database on the specified host and port:

  ```
  CONNECT WRK/db/Sports2020 -H dbserver -S 1900 NO-ERROR.
  ```

- Startup parameters — You can use startup parameters to establish a connection to one or more databases. In the following example, the `-db` startup parameter is used to specify the name of the database.

  ```
  prowin -db sports2020 -1
  ```

  For more information, see "Client startup command syntax" in *Startup Command and Parameter Reference*.

2

Concepts and Syntax

In the following topics, we introduce important ABL concepts and basic syntax. You first learn what the basic ABL data types are and how to define variables. Then you learn about statements that control program flow, such as `IF THEN ELSE`, `CASE`, `DO`, `FOR`, and `REPEAT`. Finally, you learn about expressions and operators, built-in functions, handle-based and class-based objects, and coding for portability.

For details, see the following topics:

- ABL basic syntax
- Variables and data types
- Control flow
- Expressions and operators
- Built-in functions
- Handle-based and class-based objects
- Code for portability

ABL basic syntax

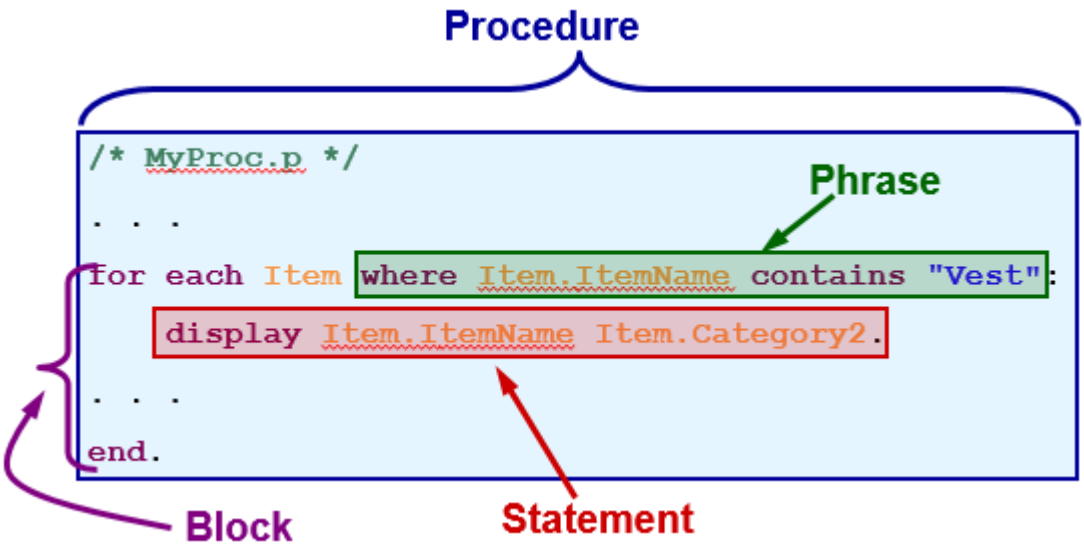
ABL code is contained in files. The files can be procedure files, class or interface files, or include files, depending on the purpose. The code within a file is made up ABL statements and blocks. Blocks are a grouping of related ABL statements that are processed as a unit. Like other programming languages you can also add comments to your code.

Procedure file

An ABL procedure (`.p`) file is a block of code that contains ABL statements. The code can be simple with a single block of code or it can be complex with multiple, nested blocks. A procedure can define data variables and also utilize input and output parameters. A procedure is compiled and executes in the ABL Virtual Machine (AVM). The compiled procedure is called an r-code (`.r`) file.

An ABL procedure (`.p`) file consists of a number of elements, including statements and blocks. It is known as an external procedure and may accept parameters or return a value.

Figure 1: ABL Programming Elements



An external procedure can directly contain code or the code can be organized into separate named entry points known as internal procedures and user-defined functions.

For more information, see An ABL procedure consists of statements in *Develop ABL Applications*, and PROCEDURE statement and FUNCTION statement in the *ABL Reference*.

Classes and interfaces

An object-oriented ABL class (`.cls`) file defines a strongly-typed class or interface. A class definition encapsulates state and behavior. A class can define data members and properties that represent the state of the class and a set of related methods and events that provide the behavior of the class. A class has one or more constructors which accept parameters, a destructor with no parameters and one or more methods that may accept parameters or return a value. The same code block structure and statements used in procedures are also used in classes.

```
CLASS foo:

  METHOD PUBLIC VOID bar ( ):
     <method-body>
  END METHOD.

  <class-body>

END CLASS.
```

An interface definition declares prototypes for a set of data members, methods, and events but does not provide the code for the methods and events. Interfaces are used to create a consistent contract among classes that implement the interface.

A class (`.cls`) file is compiled and executes in the AVM. The compiled class is also called an r-code (`.r`) file.

For more information, see Get Started with Classes, Interfaces, and Objects.

Statements

An ABL statement is one complete instruction to the AVM at runtime. Within a statement, you use several language elements such as ABL keywords, variables, and operators. A statement may also contain constants, data class members, or database field names. Keywords are used to specify the syntax for a particular type of ABL statement. Variables are placeholders for values that are assigned at runtime. Constants are typically strings (in single or double quotes), logical values (true/false or yes/no), or numbers. An ABL statement always ends with a period. The following is an example of a single statement that displays "Hello, World!" in a message alert box.

```
MESSAGE "Hello, World!" VIEW-AS ALERT-BOX.
```

Running the code produces the following output:

```
Hello, World!
```

ABL is case-insensitive, meaning you can write either an ABL keyword or a reference to a variable name, or database table or field name, in any combination of uppercase and lowercase letters. There are circumstances when working with operating system-level commands or file system names when case must be correct. It is advisable to always write case correct code.

For more information, see An ABL procedure consists of statements and ABL combines procedural, database, and user interface statements in *Develop ABL Applications.*. The ABL Syntax Reference contains reference information on all the ABL statements.

Blocks

ABL is a block-structured language. Blocks contain a set of ABL statements that are executed in sequence and are terminated by an `END` statement. The top level or outermost block of a procedure file is called the main block. ABL code blocks can be nested. Some ABL statements define blocks that begin with a block-header statement that end with a colon (:), and conclude with an `END` statement.

The following example code shows a block. Note that the `FOR EACH` statement starts the block.

```
FOR EACH Customer:
  DISPLAY CustNum Name.
END.
```

Blocks are typically used to manage transactions. A transaction is a set of changes to a database, or an undoable temp-table, variable, or class data member, which can be rolled back if something unexpected happens. Transactions are scoped to blocks and every block has default error handling built-in. When a transaction block is nested within another transaction block, it is called a subtransaction. When an error occurs, the first step is to undo the transaction or subtransaction. Thus, undoing a transaction reverses changes to database fields, undoable variables, class data members,and temp-table fields. You can further control error handling by defining explicit block error handling and by adding `CATCH` and `FINALLY` statements to the end of blocks.

Blocks are also used to create a scope for a database buffer. (You learn more about database buffers in a later topic.)

There are some general rules for determining what constitutes a block:

- ABL statements such as FOR EACH, DO, and REPEAT define the start of a new block.
- Every procedure itself constitutes a block.
- A database trigger (block of ABL code that executes whenever a specific database event occurs) is a block of its own.
- An internal procedure, user-defined function, constructor, method, and destructor define a block.

The complete list of block types is summarized in the following table:

Table 1: ABL block types

Block name	Defined by
Simple `DO` block	`DO` statement used to group multiple statements into a single execution unit
`DO TRANSACTION` block	`DO` statement with the `TRANSACTION` option
`DO ON <directive>` block	`DO` statement with a directive, such as `ON ERROR`
`FOR` block	`FOR` statement
`REPEAT` block	`REPEAT` statement
`CATCH` block	`CATCH` statement
`FINALLY` block	`FINALLY` statement
Procedure (External procedure)	The implicit all-enclosing block of a procedure file
Internal procedure	`PROCEDURE` statement
User-defined function	`FUNCTION` statement
Database trigger procedure	`TRIGGER PROCEDURE` statement in a procedure file
Database trigger block	`ON` statement with a database event specified
User-interface trigger	`ON` statement with a user-interface event specified
Class method (User-defined method)	`METHOD` statement

Block name	Defined by
Class constructor	CONSTRUCTOR statement
Class destructor	DESTRUCTOR statement
Class property	PROPERTY statement and GET or SET definition
Class block (Class file)	CLASS statement in a class file (.cls)

For more information, see Procedure Blocks and Data Access and Transactions and trigger and procedure blocks in *Develop ABL Applications*.

Comments

An ABL application can also contain non-executable comments. There are two styles of comments: // Single-line comments and /* Multi-line comments */. Single-line comments begin with // and stop at the end of the line. Multi-line comments begin with /* and end with */. This type of comment can wrap over multiple lines or be on a single line. The following example shows both comment styles:

```
/* Step through all customers with a balance under 1000 */

FOR EACH Customer NO-LOCK WHERE Customer.Balance < 1000:
  DISPLAY Customer.Name.
  FOR EACH Order OF Customer NO-LOCK:
    DISPLAY Order.OrderDate.    // Display order dates of each customer
  END.
END.
```

Variables and data types

ABL lets you define program variables for use within an application. A variable must be defined with a specific data type. A variable can be initialized with a value when it is defined, or it can be assigned a value later. You use the VAR statement or the DEFINE VARIABLE statement to define variables. The VAR statement is a newer shorthand notation for the DEFINE VARIABLE statement that is less verbose. Most variables can be defined using the VAR statement.

Define a variable using the VAR statement

The basic simplified syntax of the VAR statement is:

```
VAR datatype varname [ = expression[, expression ]...] .
```

datatype

Specifies the data type of the variable, for example, CHAR, INT, DECIMAL, DATE, LOGICAL. See the following table for a list of supported data types.

varname

Specifies the name of the variable.

expression

An Expression which evaluates to a data type that is consistent with the data type of the defined variable. See the *Default initial values* table below for a list of default initial values for each data type.

The following example code shows some variables definitions using VAR:

```
/* CHARACTER variable with an initial value */
VAR CHAR s1 = "abc".

/* INTEGER variable */
VAR INT count.

/* DATE variable with an initial value */
VAR DATE d1 = 1/1/2020.

/* LOGICAL (boolean) variable */
VAR LOGICAL isPaidFor.

/* CHARACTER array with three elements */
VAR CHAR[3] s2.

/* INTEGER array with two elements */
VAR INT[2] z = [100, 200].

/* Instantiated object variable */
VAR myclass myobj = NEW myclass("Progress", 2021, ?).

/* Determinate array object variable */
VAR StateClass[2] objArrayA = [NEW StateClass(“MA”), NEW StateClass(“NH”)].

/* Indeterminate array object variable */
VAR StateClass[] objArrayB = [NEW StateClass(“MA”), NEW StateClass(“NH”)].

/* Object variable using expressions as parameters */
VAR point p1 = NEW point(x - 5, y + 5).
```

Define a variable using the DEFINE VARIABLE statement

The basic simplified syntax of the DEFINE VARIABLE statement is:

```
DEFINE VARIABLE varname AS datatype
  [ INITIAL { value } ]
  [ EXTENT [ n ] ]
  [ NO-UNDO ].
```

varname

Specifies the name of the variable.

datatype

Specifies the data type of the variable, for example, CHARACTER, INTEGER, DECIMAL, DATE, LOGICAL. See the following table for a list of supported data types.

`INITIAL` *value*

Optionally specifies an initial value for the variable. If specified, the value must be a constant. See the following table for a list of default initial values for each data type.

`EXTENT` *n*

Optionally specifies that the variable is an array of size *n*. The size must be an integer. After the array is defined, you can then reference the individual array elements in your code by enclosing the array subscript in brackets, as in `myVar[2] = 5.`

`NO-UNDO`

Specifies that the variable does not require transaction support. The best practice is to always use `NO-UNDO` for variables unless explicitly required, to prevent unnecessary overhead.

See Other variable qualifiers in *Develop ABL Applications* for more information on other qualifiers.

Data types

ABL supports a range of Data types, and each data type has a default initial value that can be overridden. The ABL data types are listed in the following table.

Table 2: Default initial values

Data type name	Default initial value
CHARACTER or CHAR	"" (The empty string)
CLASS *object-type-name*	Unknown value (?)
DATE	Unknown value (?)
DATETIME	Unknown value (?)
DATETIME-TZ	Unknown value (?)
DECIMAL	0
HANDLE	Unknown value (?)
INTEGER or INT	0
INT64	0
LOGICAL	no
LONGCHAR	Unknown value (?)
MEMPTR	Unknown value (?)
RAW	A zero-length sequence of bytes
ROWID	Unknown value (?)

Assign a value to a variable

ABL assigns values to variables using the assignment statement (=). In the following example, the variable, *name*, is set to the string, `NewCustomer`.

```
/* Set a local string variable to "NewCustomer" */
name = "NewCustomer".
```

See Assigning a value to a variable for more information.

Compound assignment operators

ABL has compound assignment operators (+=, -=, *=, /=), which allow you to combine addition, subtraction, multiplication, and division operators with an assignment. See Compound assignment operators on page 20 for more information.

ABL Unknown value

For certain data types, when a value is not yet assigned to the data element, ABL associates the Unknown value with the data element. In ABL, you represent the Unknown value with a question mark (?). Note that you do not put quotation marks around the question mark; the question mark acts as a special symbol on its own. The Unknown value (?) is not equal to any defined value. In ABL, you write a statement that looks as though you are comparing a value to a question mark, such as: `IF ShipDate = ?`, but the statement is asking if `ShipDate` has the Unknown value, which means it has no particular value. The Unknown value is like the NULL value in SQL, and the expression `IF ShipDate = ?` is like the SQL expression `IF ShipDate IS NULL`.

For more information, see Using the ABL Unknown value.

Compound assignment operators

ABL has four compound assignment operators (+=, -=, *=, and /=), which can be used to perform an operation and assign a value, using a shorthand notation. These assignment operators make your code more concise and easier to read and understand. Supported operations include addition (numeric and date-based) and concatenation, subtraction (numeric and date-based), multiplication, and division.

The following table compares the syntax for each assignment operator with the equivalent basic syntax, and notes which types of operations are supported.

Operator	Assignment operator syntax	Equivalent basic syntax	Supported operations
+=	*field += expression*	*field = field + (expression)*	Concatenation, numeric and date-based addition
-=	*field -= expression*	*field = field - (expression)*	Numeric and date-based subtraction
*=	*field *= expression*	*field = field * (expression)*	Numeric multiplication
/=	*field /= expression*	*field = field / (expression)*	Numeric division

See the *ABL Reference* entries for more information:

- += Addition assignment operator
- -= Subtraction assignment operator
- *= Multiplication assignment operator
- /= Division assignment operator

Specify display format for variables

The DISPLAY statement is an ABL statement that can be used for displaying output. This statement can be used in conjunction with the built-in user interface (UI). Most modern applications do not use the built-in UI for showing output, but the built-in UI can be useful when prototyping or debugging your code. The built-in UI is helpful when learning ABL, because you can try ABL code and quickly display the output without having to learn another UI.

When using the `DISPLAY` statement to display the value of a variable, you can either use the ABL default display format for the data type of the variable, or you can specify a custom display format. You specify a custom format by using the `FORMAT` phrase in the DEFINE VARIABLE statement.

The simplified syntax for the `DEFINE VARIABLE` statement, with the `FORMAT` phrase, is shown:

```
DEFINE VARIABLE variable-name AS data-type
     [ FORMAT "format-phrase" ] [ NO-UNDO ].
```

In the following example code, the `DEFINE VARIABLE` statements for the `myStrCustom` and `myDecCustom` variables override the default formatting. `myStrCustom` is defined to display twenty characters. `myDecCustom` is defined to display four decimal places. For comparison purposes, two additional variables are defined, `myStrDefault` and `myDecDefault`, which do not alter the default formatting.

```
DEFINE VARIABLE myStrDefault AS CHARACTER NO-UNDO.
DEFINE VARIABLE myStrCustom AS CHARACTER FORMAT "x(20)" NO-UNDO.
DEFINE VARIABLE myDecDefault as DECIMAL NO-UNDO.
DEFINE VARIABLE myDecCustom as DECIMAL FORMAT "->>,>>9.9999" NO-UNDO.

myStrDefault = 'abcdefghijklmnopqrstuvwxyz'.
myStrCustom = 'abcdefghijklmnopqrstuvwxyz'.
myDecDefault = 123.456789.
myDecCustom = 123.456789.

DISPLAY myStrDefault myStrCustom myDecDefault myDecCustom.
```

Running the code produces the following output:

```
myStrDefault myStrCustom          myDecDefault  myDecCustom
------------ -------------------- ------------ ------------
abcdefgh     abcdefghijklmnopqrst       123.46     123.4568
```

For more information on the default display formats, see "DEFINE VARIABLE statement" in *ABL Reference*.

Note: Neither default nor custom display formats apply to the `MESSAGE` statement.

Data type processing

The following topics discuss how to work with ABL data types and describe some of the useful ABL functions that support them. ABL character lists, arrays, and handle objects are also covered.

- Numeric data
- Dates and times
- Logical (boolean) data
- Character strings
  - CHARACTER data
  - LONGCHAR data
  - Lists
- One-dimensional arrays (EXTENTS)
- Handles

Work with numeric data

ABL provides three numeric data types that you can use in your application.

ABL data type	Description
INTEGER or INT	A 32-bit integer value within the range -2147483648 to +2147483647.
INT64	A 64-bit integer value within the range -9223372036854775808 to +9223372036854775807.
DECIMAL	A signed value that holds a maximum of 50 digits, including 10 digits to the right of the decimal place.

All numeric data types have a default initial value of 0. However, you can optionally set initial values when defining variables.

Arithmetic operators

The standard arithmetic operators (`+, -, *, /, modulo`) are used to perform calculations in ABL. You can perform calculations using variables, constants, database field values, and functions.

Arithmetic operator	Name	Example
+	+ Addition operator	`dBalance = dBalance + dVarPrice`
-	- Subtraction operator	`dBalance = dBalance – dTotalPaid`
*	* Multiplication operator	`dExtendedPrice = dPrice * dQuantity`
/	/ Division operator	`dVarPrice = dExtendedPrice / iQuantity`
modulo	MODULO operator	`dQuota = iTotal MODULO 4`

Order of evaluation

The evaluation of numeric expressions in ABL follow the standard order of evaluation used in most programming languages:

1. Multiplication
2. Division
3. Modulo
4. Addition
5. Subtraction

Use parentheses for clarity

It is a best practice to use parentheses to explicitly indicate the desired evaluation order. This makes your code clear and easy to maintain, and ensures the result is what you intended.

```
dBonus =  ((dQuota / dTSales) * dBasePay) + 1000.
```

In numeric expressions, you can use the + and - operators in two ways. If there is a space between the operator and the following numeric value, then the operator is interpreted as an arithmetic operator. Otherwise, it is interpreted as a unary operator that makes the numeric value a positive or negative value. Make sure you include the space if you want an arithmetic operation.

Comparison operators

ABL provides operators to compare variables. You can use either the character representation or the symbol for each operator.

Operator
EQ or = operator
NE or <> operator
LT or < operator
LE or < = operator
GT or > operator
GE or >= operator

Convert numeric data

You should be mindful of how numeric data may be automatically converted. If you add an INTEGER with a DECIMAL then the result is converted to the DECIMAL type. If you assign a DECIMAL value to an INTEGER variable, then the result is converted to an INTEGER with rounding.

In addition, ABL provides built-in functions to convert and operate on numeric data, as shown. Note that these functions require parameters.

ABL function
ABSOLUTE
DECIMAL
EXP
INTEGER
INT64

ABL function
LOG
MAXIMUM
MINIMUM
RANDOM
ROUND
SQRT
TRUNCATE

Example using INTEGER, ROUND, and TRUNCATE

The following example code demonstrates the `INTEGER`, `ROUND`, and `TRUNCATE` functions:

```
VAR DECIMAL test = 350.9875.

MESSAGE "Original value:" test SKIP
  "Integer value:" INTEGER(test) SKIP
  "Rounded value:" ROUND(test,0) SKIP
  "Truncated value:" TRUNCATE(test,0).
```

The example code produces the following output:

```
Original value: 350.9875
Integer value: 351
Rounded value: 351
Truncated value: 350
```

Work with dates and times

You can define three types of date- and time-related data in your application:

ABL data type	Description
DATE	A value that represents a date using the Gregorian calendar.
DATETIME	A value that consists of an ABL DATE type and the time in milliseconds starting at midnight on that date.
DATETIME-TZ	A value that consists of an ABL DATETIME type and the time zone offset from Coordinated Universal Time (UTC) in minutes. The format of the time zone offset is hours and minutes.

The default initial value for these types is the Unknown value (?). If you attempt to write or display a date that was not assigned a value, then it is written or displayed as a blank.

If you share a DATE or DATETIME value with another process, you must take into consideration time zone information and how system time is calculated on different platforms.

Example date and time formats

ABL type	Format specification	Sample output
DATE	`99/99/99` (Default)	05/30/20
	`99/99/9999`	05/30/2020
	`99-99-99`	05-30-20
DATETIME	`99/99/9999 HH:MM:SS.SSS` (Default)	05/30/2020 15:30:44.234
	`99/99/9999 HH:MM`	05/30/2020 15:30
	`99-99-99 HH:MM:SS am`	05-30-2020 03:30:44 pm
DATETIME-TZ	`99/99/9999 HH:MM:SS.SSS+HH:MM` (Default)	05/30/2020 16:30:44.234-03:00
	`99/99/9999HH:MM+HH:MM`	5/30/2020 16:30-03:00

Define and initialize date and time variables

To initialize a DATE, DATETIME, or DATETIME-TZ variable, you must specify a value that matches the type of data you want to initialize. The specification of a DATETIME or DATETIME-TZ initial value that contains spaces requires quotation marks around it. You do not need to use quotation marks for the initial value of a DATE type or if specifying a DATETIME in ISO date/time format.

The following example code defines a variable of each of the ABL date and time types.

```
VAR DATE dtShipDate = 02-14-2020.
VAR DATETIME dtShipTime = "02-14-2020 14:45".
VAR DATETIME-TZ dtOrderTime = "12-17-2020 10:15-5:00".
```

Note: The date and time part of a DATETIME-TZ is stored relative to the UTC in the database/temp-table field or the variable. This occurs so it can be indexed in absolute terms. When displaying the value, the AVM converts the value to the local time relative to the time zone.

Date and time operators and functions

ABL provides operators and built-in functions for working with dates and times.

Date and time operators and functions
+ Date addition operator
- Date subtraction operator
+ Datetime addition operator
- Datetime subtraction operator
ADD-INTERVAL function
DATE function
DATETIME function
DATETIME-TZ function
DAY function
ETIME function
INTERVAL function

Date and time operators and functions
ISO-DATE function
MONTH function
MTIME function
NOW function
TIME function
TIMEZONE function
TODAY function
WEEKDAY function
YEAR function

The following example code uses several of these operators and functions:

```
SESSION:DATE-FORMAT = "mdy".
MESSAGE "Today's date is" TODAY SKIP
  "Today's datetime is" NOW SKIP
  "Tomorrow's date is" TODAY + 1 SKIP
  "Yesterday's date was" TODAY - 1 SKIP
  "The current month is" MONTH(TODAY) SKIP
  "Last month was" MONTH(TODAY) - 1 SKIP
  "Next year is" YEAR(TODAY) + 1 SKIP
  "The current datetime is" NOW SKIP
  "The current datetime in ISO-Date format is" ISO-DATE(NOW)
  VIEW-AS ALERT-BOX.
```

Running the example code produces output similar to the following:

```
Today's date is 03/25/20
Today's datetime is 03/25/2020 16:56:10.826+00:00
Tomorrow's date is 03/26/20
Yesterday's date was 03/24/20
The current month is 3
Last month was 2
Next year is 2021
The current datetime is 03/25/2020 16:56:10.826+00:00
The current datetime in ISO-Date format is 2020-03-25T16:56:10.826+00:00
```

Set the display order for dates

The default order for displaying date information is month, followed by day, followed by year (mdy). You can change the default order by specifying the Date Format (-d) startup parameter (`-d dmy` or `-d ydm`). In addition, you can temporarily change the display order for the current session by setting the `DATE-FORMAT` attribute on the `SESSION` system handle. (You learn more about handles in Work with handles on page 42.)

The following example code displays the current date (using the `TODAY` function) and date and time (using the `NOW` function). The date display order is then changed to `dmy` for the current ABL session. The current date and time are then redisplayed.

```
MESSAGE "Today's date is" TODAY SKIP
  "Today's date and time is" NOW
  VIEW-AS ALERT-BOX.

SESSION:DATE-FORMAT = "dmy".
```

```
MESSAGE "Date format changed." SKIP
  "Today's date is" TODAY SKIP
  "Today's date and time is" NOW
  VIEW-AS ALERT-BOX.
```

Running the code produces the following output:

```
Today's date is 04/25/20
Today's date and time is 04/25/2020 14:54:31.648-04:00
```

```
Date format changed.
Today's date is 25/04/20
Today's date and time is 25/04/2020 14:54:31.649-04:00
```

Calculate intervals with dates and times

You can calculate intervals between two dates and times using the `INTERVAL` function. When you use this function, the types of the two variables do not need to be the same. You provide two DATE, DATETIME, or DATETIME-TZ values, and the function subtracts the milliseconds in the second value from the milliseconds in the first value. The value returned is in the units of the interval type you requested.

The syntax for the `INTERVAL` function is shown:

```
INTERVAL(dtVal1, dtVal2, interval-type)
```

dtVal1

The first DATE, DATETIME, or DATETIME-TZ value.

dtVal2

The second DATE, DATETIME, or DATETIME-TZ value.

interval-type

Can be any of the strings: "years", "months", "weeks", "days", "hours", "minutes", "seconds", "milliseconds".

In the following example code, the `INTERVAL` function requests the number of days between `dtShipDate` and `dtOrderTime`.

```
VAR DATE dtShipDate = 02-14-2020.
VAR DATETIME-TZ dtOrderTime = "02-01-2020 10:15-5:00".

MESSAGE "Order time:" dtOrderTime SKIP
  "Ship date:" dtShipDate SKIP
  "Days to process order:" INTERVAL(dtShipDate,dtOrderTime,"days")
  VIEW-AS ALERT-BOX.
```

Running the code produces the following output:

```
Order time is: 02/01/2020 10:15:00.000-05:00
Ship date is: 02/14/20
Days to process order: 12
```

Work with logical data

Logical data represents a binary value—YES (TRUE) or NO (FALSE). You use logical values and expressions in many parts of your application. Many functions, methods, and procedures return values that report the success or failure of their execution. In your ABL code, you typically check a return value to determine how to continue execution.

When you define a logical variable, you can initialize it. The possible values are YES, NO, TRUE, or FALSE. YES is the same as TRUE, and NO is the same as FALSE. If you do not initialize the variable when you define it, then the default initial value is NO.

Logical expressions

A logical expression can contain a combination of comparison and logical operators. The comparison operators include <, >, <=, >=, and =. In addition, ABL provides the logical operators `AND`, `OR`, and `NOT`.

Operator	Example	Explanation
AND	`CreditLimit = 1000 AND Country = "USA"`	Checks whether the `CreditLimit` is `1000` and the `Country` is USA.
OR	`Country = "UK" OR Country = "USA"`	Checks whether the `Country` is either UK or USA.
NOT	`(NOT (Balance = 0)) AND Country = "USA"`	Checks whether the `Balance` is not zero and the `Country` is USA.

When using a combination of comparison and logical operators, there are often many ways to filter data using different expressions. The following examples are equivalent:

```
(NOT (Balance = 0)) AND Country = "USA"
```

```
Balance <> 0 and Country = "USA"
```

Precedence of operators

As with numeric expressions, a logical expression also has a precedence of operators. A best practice is to use parentheses to explicitly control the order of evaluation of a logical expression, just as you do for numeric expressions.

The following example code shows a logical expression that uses parentheses. This expression uses comparison operators and the logical operators `AND` and `NOT`.

```
FOR EACH Customer WHERE (NOT (Customer.CreditLimit)) > dTotal AND
       (Customer.Balance = 0):
  /* do something with the records that match the where clause */
END.
 . . .
```

Work with character strings

An ABL application can define character data by using either a CHARACTER or LONGCHAR variable. A CHARACTER variable can hold values as character strings up to 32K. If you need to store a character string that is greater than 32K, you use the LONGCHAR data type.

You can also use a LONGCHAR variable if you want a character string in a specific code page that is not the same as the internal code page (cpinternal). For more information about code pages, see "Code pages" in *Internationalize ABL Applications*.

A LONGCHAR field cannot be defined in an OpenEdge database table or temp-table. You use the Character Large Object (CLOB) data type for large strings in a database field. You can transfer a CLOB field from the database to a LONGCHAR field so that it can be manipulated or viewed with ABL code.

Work with CHARACTER data

When you define a variable as a CHARACTER type, its default initial value is an empty character string (""). You can initialize the variable to set the initial value. In ABL, you can modify the value of a CHARACTER variable, provided the result does not exceed 32K characters.

```
VAR CHAR cURLString.

VAR CHAR cWarehouseName = "Bldg A".
```

ABL provides many operators and functions for working with strings. All of these can be used with CHARACTER data, and most of these can also be used with LONGCHAR data.

String functions and operators	Description
+ Concatenation operator	Joins two character strings or expressions.
ASC function	Converts a character expression representing a single character into the corresponding ASCII (or internal code page) value, returned as an INTEGER.
BEGINS operator	Returns true if a string begins with a pattern.
CAPS function	Returns the uppercase version of a string.
CHR function	Converts an integer value to its corresponding character value.
COMPARE function	Compares two strings using a specified comparison operator or function.
FILL function	Generates a string made up of a character string that is repeated a specified number of times.
INDEX function	Searches the string for a substring, and if it is found, then returns the location of the substring.
LC function	Returns the lowercase version of a string.
LEFT-TRIM function	Removes characters from the beginning of a string, including leading blanks.
LENGTH function	Returns the length of the string.
MATCHES operator	Performs a wildcard match for a string within another string. By default MATCHES is not case sensitive.
OVERLAY statement	Inserts content from a specified expression into a field or variable replacing existing characters, bytes, or columns.

String functions and operators	Description
R-INDEX function	Returns an INTEGER value that indicates the position of the target string within the source string. In contrast to the INDEX function, R-INDEX performs the search from right to left.
REPLACE function	Returns a string with specified substring replacements. The data type of the returned value matches the data type of the expression passed to the function.
RIGHT-TRIM function	Removes characters from the end of a string, including trailing blanks.
STRING function	Converts a value of another data type into a character value.
SUBSTITUTE function	Returns a character string that is made up of a base string plus the substitution of arguments in the string.
SUBSTRING function	Returns a substring of a string based on the locations specified in the string.
TRIM function	Removes characters from the beginning or end of a string, including leading or trailing blanks.

For additional information, see ABL string manipulation functions.

The next sections discuss how to use the concatenation operator (+) and the `SUBSTITUTE`, `REPLACE`, and `STRING` functions. These are commonly used for string manipulation. See the *ABL Reference* for more information on the other functions listed in the table.

Concatenation operator (+)

You can combine character strings by concatenating them. Concatenation means putting two or more strings together in sequence. For example, the string "PRO" concatenated with the string "GRESS" becomes "PROGRESS".

You use the string + Concatenation operator to combine two or more strings:

```
string1 + string2 + string3
```

If you want spaces or punctuation to appear between string values, then you must explicitly include them. For example, to display a person's full name, you might write the following code:

```
VAR CHAR fullName.
VAR CHAR firstName = "Alexander".
VAR CHAR middleInitial = "T".
VAR CHAR lastName = "Great".

fullName = firstName + " " + middleInitial + ". " + lastName.

MESSAGE "My name is: " fullName.
```

Running the code produces the following output:

```
My name is:  Alexander T. Great
```

SUBSTITUTE function

The SUBSTITUTE function returns a character string that is made up of a base string plus the substitution of parameter values in the string. This function allows you to use a single string in place of concatenated strings.

```
new-string = SUBSTITUTE( base-string [ , arg ] ... )
```

new-string

The string returned by the SUBSTITUTE function.

base-string

A character string optionally containing substitution parameters of the form &*n*, where *n* is an integer between 1 and 9, inclusive.

arg

A constant, field name, variable, or expression that results in a character string value. These values replace substitution parameters in *base-string*.

The following example code uses substitution rather than concatenation, but produces the same output as the previous example. In this example, `firstName` replaces `&1`; `middleInitial` replaces `&2`; and `lastName` replaces `&3`.

```
VAR CHAR newString.
VAR CHAR firstName = "Alexander".
VAR CHAR middleInitial = "T".
VAR CHAR lastName = "Great".

newString = SUBSTITUTE("My name is: &1 &2. &3", firstName, middleInitial, lastName).

MESSAGE newString.
```

REPLACE function

You can use the REPLACE function to replace characters in a variable with an expression.

The syntax for the REPLACE function is:

```
new-string = REPLACE(source-string, from-string, to-string).
```

new-string

The string returned by the REPLACE function.

source-string

Specifies the base string to make replacements in. The *source-string* parameter can be any expression that evaluates to a string. The REPLACE function does not change the value of *source-string* itself, unless *new-string* and *source-string* refer to the same variable.

from-string

Specifies the substring to replace. The *from-string* parameter can be any expression that evaluates to a string. Each occurrence of *from-string* within *source-string* is replaced.

to-string

Specifies the replacement substring. The *to-string* parameter can be any expression that evaluates to a string. Each occurrence of *from-string* in *source-string* is replaced by *to-string*.

Example: Replace a text string

The following example code creates a list of departments separated by commas. The REPLACE function changes the separator from a comma to a bar surrounded by spaces, for better readability.

```
VAR CHAR cDeptList.

FOR EACH Department:
  cDeptList = cDeptList + "," + DeptName.
end.
MESSAGE cDeptList VIEW-AS ALERT-BOX.

cDeptList = REPLACE(cDeptList, ",", " | ").
MESSAGE cDeptList VIEW-AS ALERT-BOX.
```

The example code produces the following output:

```
,Consulting,Administration,Marketing,Sales,Training,Development,Finance,Shipping
```

```
 | Consulting | Administration | Marketing | Sales | Training | Development | Finance
| Shipping
```

Convert data to the ABL CHARACTER type

The STRING function allows you to convert ABL non-character data types to a character value. Many data types are supported, including:

- INTEGER or INT
- DECIMAL
- LOGICAL
- DATE
- DATETIME
- DATETIME-TZ

You can concatenate a string representing different data types, provided you first convert each non-character type to a string.

The STRING function syntax is shown:

```
STRING ( value [, "format-string" ] )
```

value

A variable, field, or an expression that evaluates to a numeric, date and time, or logical value.

format-string

The customized format for the string that is returned by this function.

Example: Use STRING() to create character data

You can create a message string from multiple values, some of which may not be CHARACTER types. In the following example code, the STRING function is used to convert a DATETIME value to a character string that is part of a concatenated message string.

```
VAR CHAR cTime.
VAR CHAR cDateStr.
VAR CHAR cTimeStr.

cDateStr = "Today's date is " + STRING(TODAY).

/* convert the current time to a string */
cTime = STRING(TIME, "hh:mm:ss am").

cTimeStr = "and the time is " + cTime.

MESSAGE cDateStr SKIP cTimeStr VIEW-AS ALERT-BOX.
```

The example code produces output similar to the following:

```
Today's date is 05/07/20
and the time is 11:20:10 am
```

Work with LONGCHAR data

A LONGCHAR variable consists of CHARACTER data, but unlike the CHARACTER type, this type is not limited to 32K in size. A LONGCHAR variable can be larger, but it may be limited by available memory. You can use many of the ABL character handling capabilities with LONGCHAR strings. The LONGCHAR data type has metadata associated with it to designate its code page and other information. The default initial value of a LONGCHAR variable is the empty string ("").

ABL has another data type known as a character large object (CLOB). A CLOB field is a database table or temp-table field that contains a locater, which points to the associated CLOB data stored in the database. You cannot manipulate CLOB data directly. Instead, you must copy a CLOB to a LONGCHAR to manipulate the character contents of a CLOB field in ABL. You can do this with the COPY-LOB statement.

Like CLOB values, LONGCHAR values can be in any code page supported by the OpenEdge convmap.p file, and the code page information is retained by the LONGCHAR. ABL ensures that all characters are valid with respect to the code page of the LONGCHAR.

Display a LONGCHAR

Because LONGCHAR variables are large in size, you cannot display the values directly. However, there are some ways around this. For shorter LONGCHAR variables, you can assign the LONGCHAR to a CHARACTER variable and then display the CHARACTER variable. Another way is to use the STRING function to convert it to a CHARACTER value. The following example code demonstrates these ways:

```
VAR LONGCHAR str1 = "abcdefghijklmnopqrstuvwxyz".
VAR CHAR str2.

MESSAGE str1.  // Results in an error

/* Assign the LONGCHAR to a CHARACTER variable */
str2 = str1.
MESSAGE str2.

/* Use the STRING function */
MESSAGE STRING(str1).
```

You can display a portion of a LONGCHAR variable using the SUBSTRING function, but you must still assign the LONGCHAR variable to a CHARACTER variable. Because the input to the SUBSTRING function is a LONGCHAR, its return value is also a LONGCHAR, and you cannot directly display a LONGCHAR. The following example code shows how to display a portion of a LONGCHAR variable.

```
VAR LONGCHAR str1 = "abcdefghijklmnopqrstuvwxyz".
VAR CHAR str2.

MESSAGE SUBSTRING(str1,1,13,"CHARACTER").  // Results in an error

/* Use the SUBSTRING function and assign it to a CHARACTER variable */
str2 = SUBSTRING(str1,1,13,"CHARACTER").
MESSAGE str2.
```

You can also display a LONGCHAR by using the DISPLAY statement with the VIEW-AS EDITOR LARGE option. The following example code shows how to display a LONGCHAR using this option. Note that you must set the number of lines and the line width, in characters. The string in the following example code does not exceed 32K, but is used for illustration purposes only.

```
VAR LONGCHAR myLongchar = "Very long string...".

DISPLAY myLongchar VIEW-AS EDITOR LARGE
  INNER-LINES 60 INNER-CHARS 100 WITH FRAME f1 WIDTH 120.
```

For more information, see the following topics in *ABL Data Types Addenda*:

- LONGCHAR data type
- Copying large objects
- What you can do with LONGCHAR values
- Restrictions
- LONGCHAR code pages

Work with lists

Comma-separated lists are used to store multiple elements in a single character variable. For example, you might want to store:

- Sales quotas for 12 months
- The department names in your company
- The names of the 12 months of the year

Define a list

A list is a string with substrings, each separated by a separator character. A comma is the default separator.

The following example code shows two examples of list variables. You do not need to initialize a list variable when it is defined, but it must have items in it in order for you to work with it.

```
VAR CHAR cMonths = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec".

VAR CHAR cQuarterlyQuotas = "100500,400000,250000,175000".
```

Access a value in a list

ABL provides functions to access and manipulate the items in a list. These functions allow you to:

- Locate the position of an item in the list (LOOKUP function).
- Extract a value from a particular position in the list (ENTRY function).
- Find out how many items are in the list (NUM-ENTRIES function).
- Set a particular item in a list to a value (ENTRY statement).

Look up an item in a list

Use the LOOKUP function to determine whether a specific item is in a list. The LOOKUP function returns 0 if the item is not in the list; otherwise, it returns an integer value of the item's position in the list.

The syntax for the LOOKUP function is:

```
position = LOOKUP(expression, list)
```

position

Returns an INTEGER value of the position of *expression* in *list*. If not found, then *position* is set to 0.

expression

A character expression to search for in *list*.

list

A CHARACTER variable that contains a comma-separated list of items.

Example: Use LOOKUP with a list

In the following example code, a list called `cMonths` is defined, which contain the months of the year. `expr` is the expression to search for in the list and is set to "Sep".

```
VAR CHAR cMonths = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec".
VAR CHAR expr = "Sep".
VAR INT position.

position = LOOKUP(expr, cMonths).

MESSAGE "cMonths:" cMonths SKIP
  "expr:" expr SKIP
  "position:" position VIEW-AS ALERT-BOX.
```

The following output is produced from running the code. Note that the position is set to 9, because "Sep" occurs in the ninth position in the list.

```
cMonths: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
expr: Sep
position: 9
```

Extract a value from a list

Use the ENTRY function to return an entry from a list based on an integer position.

```
item = ENTRY(position,list)
```

The following example code extracts address data from a comma-separated list.

```
VAR CHAR cAddress.

cAddress = "Lift Tours,276 North Drive,,Burlington,MA,01730,USA".

MESSAGE "Name:" ENTRY(1,cAddress) SKIP
  "Address line 1:" ENTRY(2,cAddress) SKIP
  "Address line 2:" ENTRY(3,cAddress) SKIP
  "City:" ENTRY(4,cAddress) SKIP
  "State:" ENTRY(5,cAddress) SKIP
  "Zipcode:" ENTRY(6,cAddress) SKIP
  "Country:" ENTRY(7,cAddress) VIEW-AS ALERT-BOX.
```

The following output is produced from running the code.

```
Name: Lift Tours
Address line 1: 276 North Drive
Address line 2:
City: Burlington
State: MA
Zipcode: 01730
Country: USA
```

Determine how many entries are in a list

Use the NUM-ENTRIES function to return how many items are in a list. `NUM-ENTRIES` is useful when you want to process items in the list in a loop. The value of `NUM-ENTRIES` tells you the number of iterations you need to do.

The example code uses NUM-ENTRIES to determine the number of times to iterate in the REPEAT block.

```
VAR INT ix = 1.
VAR CHAR cMonths = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec".

REPEAT ix = 1 TO NUM-ENTRIES(cMonths):
  DISPLAY ix " " ENTRY(ix, cMonths).
END.
```

Running the code produces the following output:

```
        ix
----------
         1   Jan
         2   Feb
         3   Mar
         4   Apr
         5   May
         6   Jun
         7   Jul
         8   Aug
         9   Sep
        10   Oct
        11   Nov
        12   Dec
```

Set a value in a list

You can substitute a value in a list with a different value by using the ENTRY statement. The syntax for the ENTRY statement is:

```
ENTRY(position, list) = expr.
```

position

An integer value that corresponds to the position of a character string in a list.

list

A list of character strings separated with a character delimiter.

expr

A character value that replaces the item in the list at the specified position.

The following example code substitutes character strings in French.

```
VAR CHAR cList = "North,South,East,West".

MESSAGE "English:" cList VIEW-AS ALERT-BOX.

ENTRY(1, cList) = "Nord".
ENTRY(2, cList) = "Sud".
ENTRY(3, cList) = "Est".
ENTRY(4, cList) = "Ouest".

MESSAGE "French:" cList VIEW-AS ALERT-BOX.
```

Running the code produces the following output:

```
English: North,South,East,West
```

```
French: Nord,Sud,Est,Ouest
```

Work with one-dimensional arrays

Arrays are used in situations where you want multiple values for a variable. In ABL, there are only one-dimensional arrays. Unlike a list, which only applies to CHARACTER data, an array can be defined for any ABL data type. To define an array, you use the VAR statement or the DEFINE VARIABLE statement. You can optionally specify the size (extent) of the array, that is the number of elements the variable can hold. If you specify a size, then the variable is a determinate array. If no size is specified when defined, then the variable is an indeterminate array and its size can be set at runtime.

Define an array variable

The syntax for defining an array variable using the VAR statement is:

```
VAR type-name[size] name [ = [val1,...valn] ].
```

The syntax for defining a variable as an array using the DEFINE VARIABLE statement is:

```
DEFINE VARIABLE name AS type-name EXTENT [size] [NO-UNDO] [INITIAL [val1,...valn]].
```

name

The name of the variable.

type-name

An ABL data type.

size

An INTEGER value containing the maximum number of elements the array can hold. If *size* is specified, then the array is considered a determinate array and its size cannot change at run time. If *size* is not specified, then the array is considered an indeterminate array and its size can change at run time.

val1,...valn

An optional, comma-separated list of initial values for the variable. The values are enclosed in brackets. You are not required to initialize all elements of the array.

Example

The following statement defines an array to hold the monthly sales quotas for a sales representative. The array is of type INTEGER and contains 12 elements:

```
VAR INTEGER[12] monthlySalesQuotas.
```

Access an element of an array

In ABL, array elements are indexed. The first element of the array has an index value of 1. Unlike lists, you cannot efficiently look up an element in an array by value. To find a specific value, you must iterate all elements using an index. To access an element of an array, you enclose the index number in brackets.

```
VAR INT[12] monthlySalesQuotas = [100500,125000,125000,175000,150000,
  155000,145000,160000,130000,140000,145000,150000].

MESSAGE "Sales quota for April: $" monthlySalesQuotas[4] VIEW-AS ALERT-BOX.
```

The example code produces the output:

```
Sales quota for April: $ 175000
```

For more information, see [ ] Array reference in the *ABL Reference*.

Determinate and indeterminate arrays

Whether the array is determinate (the size cannot change) or indeterminate (the size can change) depends on its definition.

To define a determinate array variable, specify an integer value for *size* using either the VAR or DEFINE VARIABLE statement.

To define an indeterminate array variable, do not specify a size in the variable definition statement. If initial values are provided in the definition, then the initial size of the array will be the number of values provided. If no initial values are provided, then the array remains uninitialized and its size must be set at runtime before you can work with it. The size of the indeterminate array can be set and reset at runtime using the EXTENT statement.

The example code shows variable definitions for determinate and indeterminate arrays:

```
VAR INT[3] a1.           // determinate array
VAR INT[3] a2 = [1,2,3]. // determinate array
VAR INT[] a3.            // indeterminate array
VAR INT[] a4 = [4,5,6].  // indeterminate array
```

An indeterminate array can be in one of two states: with a size (fixed) or without a size (unfixed). An indeterminate array does not have a size when first defined, unless initial values are provided. You can set the size of an indeterminate array with no size by:

- providing initial values. The size becomes the number of elements provided.
- setting the number of elements in the array using the EXTENT statement.
- assigning an array with a size to the indeterminate array. The size becomes the size of the source array.
- passing array parameters to a procedure, user-defined function, or class-based method, so that the indeterminate array value is the target for the passing of a source array with a size. The array size becomes the size of the source array.

EXTENT function

To determine how many elements are in an array, you can use the EXTENT function. A numeric value is returned if there are elements. If there are no elements, then the Unknown value (?) is returned.

```
/* Indeterminate array */
VAR INT[] monthlySalesQuotas = [100500,125000,125000,175000,150000,155000,145000,
  160000,130000,140000,145000,150000].

/* Determinate array */
VAR INT[12] monthlySales.

/* Indeterminate array */
VAR INT[] qtr1Sales.

MESSAGE "monthlySalesQuotas array has" EXTENT(monthlySalesQuotas) "elements." SKIP
  "monthlySales array has" EXTENT(monthlySales) "elements." SKIP
  "qtr1Sales array has" EXTENT(qtr1Sales) "elements."
  VIEW-AS ALERT-BOX.
```

Running the code produces the output:

```
monthlySalesQuotas array has 12 elements.
monthlySales array has 12 elements.
qtr1Sales array has ? elements.
```

EXTENT statement

The EXTENT statement sets or resets the size of an indeterminate array after it is defined. The EXTENT statement:

- Sets the number of elements for an indeterminate array variable or parameter
- Resizes an indeterminate array variable or parameter

After the size is set, you can use the indeterminate array as you would any determinate array. You cannot use the EXTENT statement on a determinate array; if you do, you get an error.

The example code uses the EXTENT statement to set the number of elements of an indeterminate array.

```
/* Define an indeterminate array */
VAR INT[] qtr1Sales.

MESSAGE "qtr1Sales array has" EXTENT(qtr1Sales) "elements."
  VIEW-AS ALERT-BOX.

/* Set the number of elements of the indeterminate array to 4 */
EXTENT (qtr1Sales) = 4.
MESSAGE "Set the size: qtr1Sales array has" EXTENT(qtr1Sales) "elements."
  VIEW-AS ALERT-BOX.

/* Resize the indeterminate array to 12 elements */
EXTENT (qtr1Sales) = 12.
MESSAGE "Resize the array: qtr1Sales array has" EXTENT(qtr1Sales) "elements."
  VIEW-AS ALERT-BOX.
```

Running the code produces the output:

```
qtr1Sales array has ? elements.
```

```
Set the size: qtr1Sales array has 4 elements.
```

```
 Resize the array: qtr1Sales array has 12 elements.
```

Assign a scalar value to all elements of an array

You can assign a scalar value to all the elements of a determinate array (or indeterminate array with a size) using the assignment operator.

```
VAR INT[2] a1. // determinate array
VAR INT[] a2. // indeterminate array

a1 = 1. // Assign 1 to each element.
DISPLAY a1.  // Display whole array (size known at compile time)

EXTENT(a2) = 2.  // Set size to 2
a2 = 2.  // Assign 2 to each element
DISPLAY a2[1] a2[2].  // Display elements individually
                      // (size not known at compile time)
```

The example code produces the output:

```
     a1[1]      a1[2]      a2[1]      a2[2]
---------- ---------- ---------- ----------
         1          1          2          2
```

Assign an array to another array (deep copy)

You can assign one array to another (Deep copying arrays), provided it makes semantic sense to do so. When deep copying one array to another, the following rules apply:

- You can assign any array to an indeterminate array whose size is not set.
- You cannot assign an indeterminate array with no size to an array with a size (determinate or indeterminate)
- If either of the arrays are a determinate array, or an indeterminate array with a size, then the size must match. If the size does not match, then the AVM raises an error.
- You cannot assign a scalar value to an indeterminate array with no size.

The example code shows a determinate array assigned to another determinate array. They both have the same size.

```
VAR INT[3] sourceArray = [1,2,3].
VAR INT[3] targetArray.

targetArray = sourceArray.

DISPLAY targetArray.
```

Running the example code produces the output:

```
targetArray[1] targetArray[2] targetArray[3]
-------------- -------------- --------------
             1              2              3
```

Work with handles

The HANDLE data type stores a pointer to a structure that represents a running built-in ABL procedure or an object in a procedure such as a socket, temp-table, or button. Handles allow you to access attributes and methods that ABL defines for each object, which you can use to interact with an object's state and behavior.

Handle-based objects are conceptually analogous to, but different from, class-based objects. Unlike class-based objects, the types and capabilities of all handle-based objects are built in to the AVM. You cannot define your own handle-based objects. Handle-based objects are also not garbage collected. You must manage the memory by cleaning up handle-based objects, using the DELETE OBJECT statement, when they are no longer needed.

ABL automatically creates instances of certain other handle-based object types in various ways, including system objects that you can access using an appropriate keyword-defined system handle. Examples of some system object handles include:

- SESSION system handle
- FILE-INFO system handle
- ERROR-STATUS system handle
- SECURITY-POLICY system handle

You define static instances, or create dynamic instances of many handle-based object types using the DEFINE and CREATE statements. You reference a static instance by the defined object name or its handle. You reference a dynamic instance only by its handle. The following example code show you how to define and access both a static and dynamic instance of an object.

```
/* Define and access a static instance of an object */

/* Define a temp-table */
DEFINE TEMP-TABLE ttCustomer NO-UNDO
  FIELD CustName AS CHARACTER
  FIELD CustId AS CHARACTER.

/* Define a handle variable */
VAR HANDLE tth.

/* Set the handle to the statically defined temp-table */
tth = TEMP-TABLE ttCustomer:HANDLE.

/* Access the HAS-RECORDS method using the handle */
DISPLAY "ttCustomer has records:" tth:HAS-RECORDS.

/* Alternatively, access the HAS-RECORDS method using the object name*/
DISPLAY "ttCustomer has records:" TEMP-TABLE ttCustomer:HAS-RECORDS.
```

```
/* Define and access a dynamic instance of an object */

/* Define a handle variable */
VAR HANDLE tth.

/* Create an empty temp-table */
CREATE TEMP-TABLE tth.
tth:ADD-NEW-FIELD("CustName","CHARACTER").
tth:ADD-NEW-FIELD("CustId","CHARACTER").
tth:TEMP-TABLE-PREPARE("cust").

/* Access the HAS-RECORDS method using the handle */
DISPLAY "tth has records:" tth:HAS-RECORDS.
```

Some commonly used object type handles in ABL are:

- Temp-table object handle
- Buffer object handle
- Buffer-field object handle
- Query object handle
- ProDataSet object handle
- Data-relation object handle
- Data-source object handle

For the complete list of system and object handles, see the Handle Reference. Attributes and methods for object type handles are described in the Handle Attributes and Methods Reference.

Control flow

ABL supports many control flow statements. In the following topics, you learn about the IF THEN ELSE, CASE, DO, FOR, and REPEAT statements.

IF ... THEN ... ELSE statement

An IF...THEN...ELSE statement is used to execute one or more ABL statements, based upon a logical condition that evaluates to true.

Syntax

```
IF expression THEN { block | statement }
  [ ELSE { block | statement } ]
```

If the value of the condition following the IF keyword is true, the AVM executes the statement or statements following the THEN keyword. If you want to execute multiple statements, they must be contained in a DO block. A DO block begins with DO: and ends with END. You can add an ELSE keyword to your IF THEN statement to handle the false condition.

The following code example contains an IF THEN ELSE statement:

```
VAR CHAR custRating.
VAR INT balance.

balance = 3000.

IF (balance <= 1000)
  THEN custRating = "A".
  ELSE DO:
       IF (balance <= 5000)
         THEN custRating = "B".
         ELSE custRating = "F".
  END.

MESSAGE "Customer has a rating of" custRating.
```

Running the code produces the following output:

```
Customer has a rating of B
```

CASE statement

The CASE statement provides a multi-branch decision based on the value of a single expression. You can specify multiple conditions, each with its own code, and execute the code when a condition is true. Each condition is defined in a WHEN … THEN section. In addition, you can define an OTHERWISE section to handle any conditions that are not explicitly defined.

Syntax

```
CASE expression :
  { WHEN value [ OR WHEN value ] ... THEN
      { block | statement }
  } ...
  [ OTHERWISE
      { block | statement }
  ]
END [ CASE ].
```

In the following example code, the CASE statement is used to assign the yearly quarter based on the month.

```
VAR CHAR qtr.

CASE MONTH(TODAY):
  WHEN 1 OR WHEN 2 OR WHEN 3 THEN qtr = "Q1".
  WHEN 4 OR WHEN 5 OR WHEN 6 THEN qtr = "Q2".
  WHEN 7 OR WHEN 8 OR WHEN 9 THEN qtr = "Q3".
  OTHERWISE qtr = "Q4".
END CASE.

MESSAGE "Today’s date is" TODAY SKIP
        "The current quarter is" qtr.
```

Running the code produces output similar to the following, depending on the current date:

```
Today’s date is 04/04/19
The current quarter is Q2
```

DO statement

The DO statement groups statements into a single block. An END statement ends the DO block.

The DO block can be a simple non-iterating block, or you can use it to iterate. There are two ways you can control iteration in a DO block:

- Iterate a specified number of times
- Iterate while a condition is true

Iterate a specified number of times

To iterate for a specified number of times, you provide a starting and ending integer value for the iteration. By default, the iteration value increments by 1 at the end of each iteration. Optionally, you can provide a different increment or decrement value using the BY clause. The iteration ends when the ending value is reached. This is the basic syntax for using DO to iterate a specified number of times:

```
DO iteration-variable = starting-value to ending-value [BY increment-value]:
  <ABL statements>
end.
```

The following is an example

```
VAR INT ix.

DO ix = 1 TO 10:
  MESSAGE "The value of ix is" ix.
END.
```

Running the code produces the following output:

```
The value of ix is 1
The value of ix is 2
The value of ix is 3
The value of ix is 4
The value of ix is 5
The value of ix is 6
The value of ix is 7
The value of ix is 8
The value of ix is 9
The value of ix is 10
```

Iterate while a condition is true

You can also iterate while a condition is true by using the `DO WHILE` construct. The following is an example:

```
VAR INT ix.

ix = 10.

DO WHILE ix > 0:
  MESSAGE "The value of ix is" ix.
  ix = ix - 1.
END.
```

Running the code produces the following output:

```
The value of ix is 10
The value of ix is 9
The value of ix is 8
The value of ix is 7
The value of ix is 6
The value of ix is 5
The value of ix is 4
The value of ix is 3
The value of ix is 2
The value of ix is 1
```

FOR statement

The FOR statement is similar to `DO`, but is specifically for iterating through records. The `FOR` statement starts an iterating block that reads a record from one or more tables at the start of each block iteration. The `END` statement terminates the `FOR` block.

Syntax

```
FOR ... :
  /* ABL statements */
END.
```

The following example uses a FOR EACH block to iterate through the records in the Customer table:

```
/* Display customers with a balance under $1000 */

FOR EACH Customer WHERE Customer.Balance < 1000:
  DISPLAY Customer.Name.
END.
```

For more information, see FOR EACH CUSTOMER in *Develop ABL Applications*.

REPEAT statement

The REPEAT statement begins a block of ABL statements that are processed repeatedly until the block ends in one of several possible ways. You use an END statement to define the end of the block. REPEAT blocks can be explicitly terminated when runtime conditions, that you have specified in your code, occur. It is good programming practice to include a terminating condition to prevent infinite loops. REPEAT blocks are also useful for iterating over records.

Syntax

```
REPEAT ... :
  /* ABL statements */
END.
```

In the following example the WHILE phrase causes the REPEAT block to terminate after 100 iterations.

```
VAR INT ix = 0.

REPEAT WHILE ix < 100:
  ix = ix + 1.
  DISPLAY ix.
END.
```

Running the code produces the following output (some output omitted for brevity):

```
        ix
----------
         1
         2
         3
         4
         5
       ...
        99
       100
```

In the following code a REPEAT block is used to iterate through database records.

```
REPEAT:
  FIND NEXT Customer.
  DISPLAY Customer.Name Customer.Balance.
END.
```

In the following example the REPEAT block is explicitly terminated:

```
REPEAT:
  FIND NEXT Customer.
  IF Customer.CustNum > 1000 THEN LEAVE.
  DISPLAY Customer.Name Customer.CustNum Customer.Balance.
END.
```

Expressions and operators

An ABL expression is a combination of one or more terms and operators that evaluate to a single value of a specific data type. An expression can include constants, variable names, database field names, or other expressions.

ABL has a set of operators for working with numeric, string, date, and logical data. The data type of the data must be compatible with the operator. One very useful operator is the concatenation operator (+), used to join two character strings or expressions:

```
FOR EACH Customer:
  DISPLAY Customer.Name Customer.City + ", " + Customer.State FORMAT "X(20)".
END.
```

More information on the ABL operators can be found in the ABL Syntax Reference.

Built-in functions

ABL supports a lengthy list of built-in functions which provide the ABL developer with easy access to a specific set of capabilities. Each function accepts a well-defined list of parameters and optionally returns a value. There are many functions available for working with strings, numbers, dates and times, and other data. The ABL Syntax Reference contains reference entries for each of these functions.

The following example demonstrates calling a function called SUBSTRING to extract a portion of a string.

```
/* Display the first three characters of the customer's name */

FOR EACH Customer:
  DISPLAY SUBSTRING(Customer.Name, 1, 3).
END.
```

Another useful function is the IF...THEN...ELSE function, not to be confused with the IF...THEN...ELSE statement. This function does not look like a typical function but can be very useful in the right situation. The function evaluates and returns one of two expressions, depending on the value of a specified condition. The following example demonstrates the IF...THEN...ELSE function.

```
VAR INT i.

i = 2.

MESSAGE IF i EQ 1 THEN "low" ELSE "high" VIEW-AS ALERT-BOX.
```

Running the code produces the following output:

```
high
```

For more information, see Using built-In ABL functions in *Develop ABL Applications*.

Handle-based and class-based objects

ABL supports two models for managing objects which provide access to a variety of capabilities:

- Handle-based objects
- Class-based objects

Handle-based objects

Handle-based objects represent built-in object types in ABL that you reference using weakly-typed object handles. These objects provide access to a variety of ABL capabilities, via their attributes and methods. You can define static or create dynamic instances of many handle-based object types. For example, the following program creates a socket object, attempts to connect to an identified resource and then deletes the object.

```
VAR HANDLE hSocket.

/* Create socket and connect to server */

CREATE SOCKET hSocket.

hSocket:CONNECT('-H localhost -S 23456') NO-ERROR.
IF hSocket:CONNECTED() = FALSE THEN
    DO:
        MESSAGE 'Unable to Connect' VIEW-AS ALERT-BOX.
        RETURN.
    END.

DELETE OBJECT hSocket.
```

In the example code you define a variable to be of type HANDLE. Then you create a socket object using the handle variable with the CREATE SOCKET statement. After creation, attributes and methods can then be called using the handle variable name, followed by a colon, and then the method or attribute name. In the example code, CONNECT and CONNECTED are methods that are called. After the object is no longer needed, DELETE OBJECT is called to clean up resources. Handle-based objects are not garbage collected. It is the responsibility of the application to delete them.

You can also chain handle references together to simplify coding. For more information, see "Chained handle references" in *ABL Reference*.

For more detail, see the ABL handle-based object reference guides:

- Handle Reference
- Handle Attributes and Methods Reference

Class-based objects

Class-based objects also represent built-in object types in ABL which you reference using strongly-typed object handles. These objects provide access to a variety of capabilities, via their properties and methods. You can define static or create dynamic instances of class-based object types.

The following program creates a `FileInputStream` object and reads the first 50 bytes from file `sample.data`. The AVM deletes this object (via garbage collection) when no more references to the object exist.

```
USING Progress.IO.*.
VAR CLASS FileInputStream rfile.
VAR MEMPTR mData.

/* Create an instance of a FileInputStream object */
rFile = New FileInputStream ("sample.data").

rFile:Read(mData, 0, 50).
rFile:Close( ).
```

For more detail, see the ABL class-based object reference guides:

- Class, Interface, and Enumeration Reference
- Class Properties and Methods Reference

Code for portability

Different operating systems have different restrictions and naming conventions. To ensure portability across platforms, take care to follow conventions when naming your fields, files, tables, and variables. The following are some recommended practices.

- Do not use a hyphen as the first character of a filename and do not use spaces in filenames.
- Use lowercase when specifying a procedure name in a `RUN` statement, and make sure your procedure files have lowercase names on UNIX.
- Use forward slashes (/) as separators in specifying file paths when using ABL statements such as `RUN`, `INPUT FROM`, or `OUTPUT TO`. The AVM automatically converts the UNIX pathname syntax to Windows pathname syntax, which helps maintain portability among operating systems.

See Naming conventions for files, tables, and variables for a complete list of conventions to follow to ensure portability across platforms.

3

Procedures and User-defined Functions

In the following topics, you learn how to work with ABL procedures and user-defined functions. You learn the difference between external and internal procedures, how to run a procedure, how to return a value from a procedure, and use parameters with procedures. You also learn about persistent procedures and how best to organize your procedure files. User-defined functions are also introduced.

For details, see the following topics:

- ABL procedures
- User-defined functions

ABL procedures

An ABL procedure file is a text file, with a `.p` extension, that contains source code for application procedures. The file itself is known as an external procedure, but it may contain within it other procedures, known as internal procedures. This topic contains information pertaining to ABL procedures.

- External and internal procedures
- Run a procedure
- Return a value from a procedure
- Use parameters with procedures
- Persistent procedures
- Organize procedure files

External and internal procedures

An ABL procedure (`.p`) file is known as an external procedure. It may accept parameters or return a string value. The following example code shows an external procedure, `helloworld.p`:

```
/* helloworld.p */

MESSAGE "Hello World".
```

An ABL internal procedure is another type of procedure that is defined by named entry points within an external procedure file. You define the start of an internal procedure using the PROCEDURE statement and indicate the end of the procedure using the END statement. You can have many internal procedures within the same external procedure file. The following example code has two internal procedures named `proc1` and `proc2` in the external procedure file, `myprocedures.p`.

```
/* myprocedures.p */

PROCEDURE proc1:
  /* ABL code */
  MESSAGE "In proc1".
END PROCEDURE.

PROCEDURE proc2:
  /* ABL code */
  MESSAGE "In proc2".
END PROCEDURE.

RUN proc1.
RUN proc2.
```

Scope of variables for internal procedures

You can define variables at a global level in the main block that can also be used by the internal procedures defined in the same file. Variables defined in the internal procedure, however, can only be used within the internal procedure.

Run a procedure

If you are running an external procedure file like `helloworld.p`, then you include the `.p` extension in the procedure name. You run internal procedures in almost exactly the same way that you run external procedures, except that there is no `.p` filename extension on the internal procedure name. The following example code shows running both an external and internal procedure:

```
/* myprocedures.p */

PROCEDURE proc1:
  /* ABL code */
  MESSAGE "In proc1".
END PROCEDURE.

PROCEDURE proc2:
  /* ABL code */
  MESSAGE "In proc2".
END PROCEDURE.

RUN helloworld.p.   // run external procedure
RUN proc1.     // run internal procedure
```

For more information, see Running ABL procedures in *Develop ABL Applications*:

Return a value from a procedure

You can use the RETURN statement to return execution to the caller, optionally specifying a string value to be returned. (Note that if you want to return a value other than a string, you need to use a FUNCTION, not a procedure.) In the caller you use the RETURN-VALUE function to retrieve the return value. Multiple RETURN statements within the same procedure are allowed. The following example code shows a return value being set in proc1, and the value retrieved in the caller:

```
/* myprocedures.p */

PROCEDURE proc1:
  /* ABL code */
  MESSAGE "In proc1".
  RETURN "1".

END PROCEDURE.

PROCEDURE proc2:
  /* ABL code */
  MESSAGE "In proc2".
END PROCEDURE.

RUN proc1.
MESSAGE "The return value from proc1 is" RETURN-VALUE.
```

For more information, see RETURN statement and RETURN-VALUE in *Develop ABL Applications*.

Use parameters with procedures

To pass values in or out of a procedure you can define parameters using the DEFINE PARAMETER statement. The parameter definition names the parameter, specifies whether it receives input or provide output, or both, and specifies the data type. This is the simplified syntax for the DEFINE PARAMETER statement:

```
DEFINE { INPUT | OUTPUT | INPUT-OUTPUT } PARAMETER parameter-name AS datatype [
NO-UNDO ] [ INITIAL initial-value ]
```

INPUT | OUTPUT | INPUT-OUTPUT

- INPUT - value is passed in to the procedure you are running.
- OUTPUT - value is returned to the caller when the called procedure completes.
- INPUT-OUTPUT - value is passed in to the procedure, which can modify the value. The AVM passes the value back to the caller when the procedure ends.

parameter-name

The name of the parameter.

datatype

The datatype of the parameter. You can use any one of the standard ABL data types, including CHARACTER, INTEGER, DECIMAL, LOGICAL, DATE. You can also specify TABLE, DATASET, TABLE-HANDLE, or DATASET-HANDLE, but these use a slightly different syntax.

initial-value

The initial value of the parameter. This only applies to OUTPUT parameters.

You specify parameters in the RUN statement to pass a parameter from the calling procedure to the called procedure. When you specify the parameter, you also specify the kind of parameter you are using, (`INPUT`, `OUTPUT`, or `INPUT-OUTPUT`). `INPUT` is the default, but a best practice is to specify the kind of parameter so that your code is unambiguous to other developers.

You must ensure that when you call a procedure that takes parameters, the number and order of parameters match, as well as their ABL data types, and input/output types. In the calling procedure, when you use variables to pass values to and from a procedure, the names of the variables need not match the names used in the procedure.

For more details, see "Parameter definition syntax" and "Parameter passing syntax" in the *ABL Reference* guide.

The following example code demonstrates passing parameters:

```
VAR CHAR v1 = "XXX".
VAR CHAR v2 = "YYY".
VAR CHAR v3 = "ZZZ".

/* proc2 */
PROCEDURE proc2:
  DEFINE INPUT PARAMETER p1 AS CHARACTER NO-UNDO.
  DEFINE OUTPUT PARAMETER p2 AS CHARACTER NO-UNDO.
  DEFINE INPUT-OUTPUT PARAMETER p3 AS CHARACTER NO-UNDO.

  p1 = "XXXXXX".
  p2 = "YYYYYY".
  p3 = "ZZZZZZ".

  MESSAGE "In proc2: p1, p2, p3:" p1 p2 p3.
END PROCEDURE.

/* main */
MESSAGE "Before running proc2: v1, v2, v3:" v1 v2 v3.

RUN proc2 (INPUT v1, OUTPUT v2, INPUT-OUTPUT v3).

MESSAGE "After running proc2: v1, v2, v3:" v1 v2 v3.
```

The following output is produced from the example code:

```
Before running proc2: v1, v2, v3: XXX YYY ZZZ
In proc2: p1, p2, p3: XXXXXX YYYYYY ZZZZZZ
After running proc2: v1, v2, v3: XXX YYYYYY ZZZZZZ
```

Persistent procedures

A persistent procedure is an instance of a procedure that stays resident in the AVM's memory until it is explicitly deleted. You can use persistent procedures as a “memory cache” of frequently used procedures and functions. The procedure maintains its state. For example, if an internal procedure you call sets a variable at the `.p` (main block) level, that value stays set and can then be used from a different internal procedure.

Since the procedure stays in memory, the AVM does not have to load the program into memory each time it is called. Thus, there is a performance benefit to your application.

To instantiate a persistent procedure in memory, you use the RUN statement with the keyword `PERSISTENT`. You also define a handle to the procedure so that you access it later.

To instantiate the persistent procedure, use the following syntax:

```
RUN proc-name PERSISTENT SET proc-handle-name.
```

In the following example code, we define the procedure handle variable, hEmpLibrary and instantiate the persistent procedure setting the procedure handle variable. You must ensure that the procedure you run is in your PROPATH at runtime.

```
VAR HANDLE hEmpLibrary.
RUN emplibrary.p PERSISTENT SET hEmpLibrary.
```

Once the persistent procedure has been instantiated, you can run any of its internal procedures in the library. You use the procedure handle to reference its internal procedures.

```
RUN proc-name IN proc-handle-name [ (parameter-list) ].
```

In the following example code, we define the procedure handle variable, hEmpLibrary and instantiate the persistent procedure in the RUN statement. In the FOR EACH statement, we call the internal procedure calcvacation for every employee and display the information.

```
/* eCheckEmpVacAvailable.p */

VAR DATE dtStart.
VAR DATE dtEnd.
VAR LOGICAL lOK.
VAR HANDLE hEmpLibrary.

RUN emplibrary.p PERSISTENT SET hEmpLibrary.


FOR EACH Employee:
  lOK = FALSE.
  RUN calcvacation IN hEmpLibrary(INPUT EMPLOYEE.EMPNUM, INPUT dtStart, INPUT dtEnd,
OUTPUT lOK).

  DISPLAY Employee.FirstName Employee.LastName lOK label "Vac?".
END.
```

Organize procedure files

There are multiple ways of organizing your procedure files. A good practice is to create procedure files of related functionality (libraries), which contain just internal procedures. You can then make these libraries available to other parts of your application. This promotes re-usability and makes it easier to maintain your application.

Another good practice is to separate user interface (UI) logic from business logic. The UI logic could be a mobile or web app, or an ABL GUI desktop client. Separating UI logic from business logic makes it easier to update the application because, in general, changes to one side of the application do not affect the other side. In a modern application, the UI logic does not directly access the database. The UI logic requests data from the business logic, which in turn, retrieves the data from the database. Exchanging data between the UI logic and the business logic is done using ABL temp-tables and datasets.

User-defined functions

A user-defined function is a block of custom code in a procedure file that returns a value. It is similar to the built-in ABL functions. It is like an internal procedure, but with the requirement that it must return a value. A user-defined function can also define parameters that are used as input, output, or both.

Before you can use a user-defined function in your code, you must define it. The definition must precede the code where it is used. Here is the simplified syntax:

```
FUNCTION  function-name RETURNS type [ ( parameter-list ) ] :
  /* ABL code */
  RETURN return-value.
END FUNCTION.
```

`function-name`

The name of the function.

`type`

The type of the return value.

`parameter-list`

Parameters are optional. A valid specification of parameters includes the parameter use (`INPUT`, `OUTPUT`, `INPUT-OUTPUT`), its name, and its type.

`return-value`

The return value of the function. The value must match the type specified in *`type`*.

In the following example code we define two functions, `OrderPrefix()` and `GeneratePO()`. The function, `OrderPrefix()`, takes no parameters and returns the prefix for the purchase order (PO) as a CHARACTER type. The `GeneratePO()` function has two parameters and constructs a PO number using these parameters.

```
/* eCustomFunctions.p */

FUNCTION OrderPrefix RETURNS CHARACTER():
  RETURN "PO".
END FUNCTION.

FUNCTION GeneratePO RETURNS CHARACTER (INPUT pcString AS CHARACTER, INPUT-OUTPUT piNumber
 AS INTEGER):
  VAR CHAR cResult.
  cResult = pcString + STRING(100 * piNumber).
  piNumber = piNumber + 1.
  RETURN cResult.
END FUNCTION.

DEFINE VARIABLE iNum AS INTEGER NO-UNDO INITIAL 999.

/* main procedure code */
MESSAGE "Purchase Order number is:" GeneratePO(OrderPrefix(),INPUT-OUTPUT iNum) SKIP
  "New iNum is: " iNum VIEW-AS ALERT-BOX.
```

You can pass a function as a parameter to another function. In the example code, you see the `OrderPrefix()` function being passed as the first parameter to the `GeneratePO()` function. Running the code produces the following output:

```
Purchase Order number is: PO99900
New iNum is:  1000
```

4

Compile and Run

In the following topics, you learn what r-code is and how to compile and run your code.

For details, see the following topics:

- R-code and the AVM
- COMPILE statement
- RUN statement
- Run using the command line

R-code and the AVM

R-code is the intermediate language to which ABL source is compiled. Once ABL has been converted to r-code, it can be run across different platforms using the ABL Virtual Machine (AVM). The AVM is the environment where ABL applications are executed. The AVM has its own directory path list, called the PROPATH, to define the directories to search and the search order when running ABL code. For more information, see "R-code Features and Functions" in*Manage ABL Applications*.

COMPILE statement

The COMPILE statement compiles a procedure file or a class definition file into r-code. A compilation can be kept in memory during a session, or you can save it permanently for use in later sessions (as an r-code file, which has a `.r` extension). When you compile a class definition file, the AVM compiles the class definition file identified in the compile statement and all class files in its inherited class hierarchy, by default.

Basic usage

```
COMPILE helloworld.p SAVE.
```

Running the code example produces a file named `helloworld.r`.

Set compiler options

You can change the behavior of the compiler, and enforce certain rules, by specifying compiler options. Using compiler options allows you to identify and fix issues in your code, making your code unambiguous and more robust. Rules you can specify include requiring full table and field names, fully qualified buffer references, full keywords, and return values for user-defined functions, non-VOID methods, and property getters. You can optionally specify a severity level (warning or error) for each type of option.

Option name	Description
require-full-names	All table and field names must appear as they are in the schema. The compiler's ability to implicitly resolve abbreviated table names is disabled.
require-field-qualifiers	All buffer references (including database tables, temp-tables, and buffers) must be fully qualified. The compiler's ability to implicitly resolve the buffer to which a field reference refers is disabled.
require-full-keywords	All ABL language keywords must be fully spelled out.
require-return-values	In user-defined functions, non-VOID methods, and property getters, all logical code paths must have `RETURN` *`value`* statements. For more information, see Require-return-values compiler option on page 61.

Setting the severity level for a compiler option

There are two possible severity levels which can be set by the user: `warning` or `error`. The `warning` level produces a message for each violation, but allows the compilation to complete successfully, provided that the only violations are related to the option. The `error` severity level produces a message for each violation and prevents the compilation from completing successfully. If you specify an option without a severity level, the default is `warning`. If you don't specify a compiler option, then no violation checking is performed for that option.

To set a severity level, append a colon (`:`) to the option name, followed by the severity level:

```
option-name:severity-level
```

`option-name`

Valid values are `require-full-names`, `require-field-qualifiers`, `require-full-keywords`, or `require-return-values`.

severity-level

Valid values are `warning` or `error`.

For example, the following are possible ways to specify severity levels for the `require-return-values` compiler option:

```
require-return-values
```

```
require-return-values:warning
```

```
require-return-values:error
```

Ways to set compiler options

There are four different ways to set compiler options, which are described below. The first two ways use the COMPILE statement. The third way uses the COMPILER system handle, and the fourth way uses the -compileroptionsfile startup parameter.

1. The OPTIONS phrase on the COMPILE statement.

   When compiling a file in ABL, use the OPTIONS phrase to specify the desired compiler options. Separate each option with a comma, and enclose the list in quotes. For example:

```
COMPILE basic.p OPTIONS "require-full-names:error,require-field-qualifiers:error".
```

   For more information on the OPTIONS phrase, see the "COMPILE statement" in *ABL Reference*.

2. The OPTIONS-FILE phrase on the COMPILE statement.

   When compiling a file in ABL, use the OPTIONS-FILE *filename* phrase on the COMPILE statement to specify the name of a file that contains the options. *filename* should be an absolute path or a path relative to PROPATH. For example:

```
COMPILE basic.p OPTIONS-FILE myCompilerOptions.
```

   The options file is a plain text file. Anything between a hash tag (#) and the end of line in the file is treated as a comment. The compiler options must be separated by commas even if on different lines. For example:

```
# myCompilerOptions
require-full-keywords, # no severity specified; warning is the default
require-full-names:error,
require-field-qualifiers:error,
require-return-values:warning
```

   For more information on the OPTIONS-FILE phrase, see the "COMPILE statement" in *ABL Reference*.

3. The OPTIONS attribute on the COMPILER system handle.

   Before compiling a file in ABL, set the OPTIONS attribute on the COMPILER system handle. For example:

```
COMPILER:OPTIONS = "require-return-values:warning".
COMPILE basic.p.
COMPILER:OPTIONS = "".   // reset to no compiler options
```

4. The -compileroptionsfile startup parameter.

   When starting up the ABL client, use the -compileroptionsfile startup parameter, to specify a file containing the options. This sets the OPTIONS attribute to the values in the file.

```
-compileroptionsfile filename
```

   The file is the same format as described in the second method for the OPTIONS-FILE phrase of the COMPILE statement. For more information on this startup parameter, see "Compiler Options File (-compileroptionsfile)" in *Open Edge Start-up Command and Parameter Referemce*.

Require-return-values compiler option

The `require-return-values` compiler option helps developers identify and fix problems, resulting in more robust code. The option helps ensure all logical code paths have RETURN *value* statements in ABL user-defined functions, non-VOID methods, and property getters. Missing RETURN *value* statements may lead to incorrect behavior and code that is hard to debug. Enabling the option during code development can flag potential issues and prevent problems down the road. The option may also be useful during troubleshooting to identify fragile code that does not behave as expected.

Enabling the option and identifying violations

To enable the feature, use the `require-return-values` compiler option. For information on enabling compiler options, see Set compiler options.

During compilation, if the option is enabled and a violation is detected, a message is displayed:

```
All code paths in user-defined function, method, or property getter "name" must return
 a value due to the "require-return-values" compiler option. (19822)
```

The message identifies the name of the user-defined function, method, or property getter with the violation. It does not identify which path or line is missing the RETURN *value* statement. You need to examine your code to determine how to make the appropriate fixes.

User-defined function, non-VOID method, or property getter block

If there are no control flow statements within the user-defined function, non-VOID method, or property getter, then the main block of the user-defined function, non-VOID method, or property getter requires a RETURN *value* statement. The following example shows a user-defined function that is not compliant with the `require-return-values` compiler option:

```
/* basic.p */
/* Not compliant with require-return-values */

FUNCTION retint RETURNS INTEGER ():

    MESSAGE "function retint".

END FUNCTION.

MESSAGE "function return" STRING(retint()).
```

Add a RETURN *value* statement to clear the warning/error, as shown in the following example:

```
/* basic.p */
/* Compliant with require-return-values */

FUNCTION retint RETURNS INTEGER ():

    MESSAGE "function retint".
    RETURN 1.

END FUNCTION.

MESSAGE "function return" STRING(retint()).
```

As long as the user-defined function, non-VOID method, or property getter ends with a RETURN *value* statement, then the code is compliant. If it does not end with a RETURN *value* statement, then you must ensure all other code paths are compliant.

Ensuring all other code paths provide a RETURN *value* statement involves control flow analysis of ABL programs.

Ensuring other code paths are compliant

As stated previously, you simply need to end each user-defined function, non-VOID method, or property getter with a RETURN *value* statement in order to make your code compliant. However, if this is not desired, then you can add RETURN *value* statements to other constructs. The following sections describe different statements that alter the flow from within ABL and how the `require-return-values` compiler option affects them. Before and after code examples are shown, along with information on how to make the code compliant. The assumption is that the containing user-defined function, method, or property getter does not end with a RETURN *value* statement, so other statements and blocks must be made compliant instead.

IF…THEN…ELSE statement

In an IF...THEN...ELSE statement, both the IF and ELSE portions require a RETURN *value* statement. Having one within either, but not both, never satisfies the `require-return-values` option. If you only have an IF clause, then you must add an ELSE clause and include a RETURN *value* statement for both.

```
/* Not compliant with require-return-values */
VAR LOGICAL boolVar = FALSE.

IF boolVar THEN
    RETURN 1.
```

```
/* Compliant with require-return-values */
VAR LOGICAL boolVar = FALSE.

IF boolVar THEN
    RETURN 1.
ELSE
    RETURN 0.
```

CASE statement

In a CASE statement, there must be an OTHERWISE clause that includes a RETURN *value* statement, and all WHEN..THEN clauses must also contain a RETURN *value* statement.

```
/* Not compliant with require-return-values */
VAR INT casetest = 3.

CASE casetest:
    WHEN 1 THEN
        RETURN 2.
    WHEN 2 THEN
        RETURN 4.
END CASE.
```

```
/* Compliant with require-return-values */
VAR INT casetest = 3.

CASE casetest:
    WHEN 1 THEN
        RETURN 2.
    WHEN 2 THEN
        RETURN 4.
    OTHERWISE
```

```
            RETURN 0.
    END CASE.
```

Nested control flow statements

Control flow statements can appear within each other. As long as each inner statement or block is compliant, then the outer statement or block is also compliant. To satisfy the `require-return-values` compiler option, it must be the case that there are no statements after the block.

CATCH and FINALLY blocks

Enforcement of the required `RETURN` *`value`* statement is not done in either a `CATCH` or `FINALLY` block. The best practice is to not use a `RETURN` *`value`* in a `CATCH` or `FINALLY` block.

RUN statement

Once you are running an AVM session, you can run a procedure from within another ABL procedure, using the RUN statement.

If you have not already created a `.r` file for the `.p` you are running, the AVM compiles it for you automatically and saves the r-code in memory. However, once the session is over, the r-code is no longer available and would have to be compiled again the next time the `.p` is run.

Basic usage

```
RUN helloworld.p.
```

Run using the command line

OpenEdge provides the `Proenv` utility to run OpenEdge command line tools. To run a procedure from the command line, use the Startup Procedure (-p) parameter. This starts a new AVM and runs the file specified by the `-p` parameter.

Basic usage

```
prowin -p helloworld.p.
```

See Use the Proenv utility for more detail.

5

Work with the OpenEdge Database

In the following topics, you learn important concepts and ABL language constructs for working with the OpenEdge database. You first learn how to access tables and buffers. Then you learn how to construct queries to limit the data retrieved. You also learn how to manage transactions and how to handle record locking. Finally you learn how to update records.

For details, see the following topics:

- Tables and buffers
- Data access
- Queries
- Transactions
- Record locks
- Update records
- Online database changes using ABL

Tables and buffers

Tables

A database table is a collection of logically related data treated as a unit. Tables contain rows and columns. All rows in a table comprise the same set of columns (fields). Tables often have indexes.

The OpenEdge database follows the relational model which organizes data into tables. Relationships between tables are defined by the data, that is, matching values in a field common to both tables. See Elements of a relational database for more detail.

In ABL, you use database access statements such as FIND statements, FOR EACH blocks, REPEAT FOR blocks, or DO FOR blocks to access the records in a table. If a database table name is specified in any of these statements, a record buffer for the table is automatically created and is populated with the current record's data.

When you retrieve a record from the database the AVM keeps track of the current record position using an index cursor—a pointer to the record.

Buffers

Whenever you reference database tables using a database access statement, Record buffers for the table are created automatically for your convenience. The record buffer is a temporary storage area in memory where the AVM manages records as they pass between the database and the statements in your code. This default record buffer has the same name as the database table. This lets you think in terms of accessing database records directly because the name of the buffer is the name of the table the record comes from.

You can create additional buffers for a table when you need to have two or more different records from the same table available to your code at the same time. For example, when looking for duplicate records, you can read a record into the default buffer and then use another buffer to go through the rest of the records for the table. If one matches then you know you have a duplicate.

Data access

ABL has four statements you can use to define a set of one or more database records: FIND, FOR, REPEAT, and OPEN QUERY. FIND fetches a single record when the statement is called. FOR , REPEAT, and OPEN QUERY identify a set of records. You can then loop through the records one at a time.

In this topic, we introduce FIND, FOR, and REPEAT. You learn about OPEN QUERY in the next topic on queries. In this topic, you also learn about ROWID, a unique internal identifier for records.

FIND statement

The FIND statement is a very powerful way to retrieve individual records from the database without having to set up a query or result set definition. FIND fetches a single record from a database table and moves that record into a record buffer. The basic (simplified) syntax for the FIND statement is:

```
FIND [ FIRST | LAST | NEXT | PREV ] record
     [ WHERE expression ]
```

`record`

Database table name.

`FIRST | LAST | NEXT | PREV`

Finds the first, last, next or previous record in the table that meets the specified characteristics.

`WHERE expression`

Restricts the query to only those rows in the database table that match the specified expression.

In the following example code, the first record from the Customer table, whose name starts with "A", is retrieved:

```
FIND FIRST Customer WHERE Customer.Name BEGINS "A".
DISPLAY Customer.Name.
```

For more information, see Data access without looping: the FIND statement in *Develop ABL Applications*.

FOR block

The FOR block iterates through a set of related records in the database table and moves each one in turn into the record buffer. The block iterates through all records that match the specified criteria and for as many iterations as requested. The FOR statement is very powerful and includes options for sorting, collating, and record-locking. When the block begins, the AVM evaluates the expression and retrieves the first record that satisfies it. This record is scoped to the entire block. Each time the block iterates, the AVM retrieves the next matching record and makes it available to the rest of the block. When the set of matching records is exhausted, the AVM automatically terminates the block. You don't have to add any checks or special syntax to exit the block at this point. The FOR block ends with an END statement. The basic (simplified) syntax of the FOR block is:

```
FOR [ EACH | FIRST | LAST ] record [ WHERE expression ]:
  /* ABL statements */
END.
```

`EACH | FIRST | LAST`

- EACH starts an iterating block, finding a single record on each iteration.
- FIRST finds the first record.
- LAST finds the last record.

`record`

Database table name.

`WHERE expression`

Restricts the query to only those rows in the database table that match the specified expression.

The following `FOR EACH` block displays the `Customer.Name` field for every record in the Customer table where the `Customer.CustNum` field is less than 100:

```
FOR EACH Customer WHERE Customer.CustNum < 100:
  DISPLAY Customer.Name.
END.
```

REPEAT block

The REPEAT block is a set of statements that are processed repeatedly but it does not automatically read records as it iterates. This block lets you navigate through a set of records yourself, rather than simply proceeding to the next record automatically on each iteration. The REPEAT block ends with an END statement.

Often a REPEAT statement statement is used with a PRESELECT phrase to select the records that meet the criteria you specify. PRESELECT creates a result list of ROWID (unique identifier) values, so that records are then retrieved by ROWID.

Typically a FIND statement is used within a REPEAT (PRESELECT) block to read a record on each iteration. There are some best practices to follow when using a REPEAT block:

1. Use the NO-ERROR qualifier on the FIND statement. This suppresses the error message that you would ordinarily get when you are at the last record.
2. Use the AVAILABLE function to check for the presence of a record. Provide a matching ELSE statement to LEAVE the block when there is no record available.

The following example code uses a REPEAT block to loop through the customer records where the Country is "USA". When the last matching record is read, the LEAVE statement breaks out of the REPEAT block.

```
FIND FIRST Customer NO-LOCK WHERE Customer.Country = "USA".
DISPLAY Customer.CustNum Customer.Name Customer.Country.

REPEAT:
  FIND NEXT Customer WHERE Customer.Country = "USA" NO-LOCK NO-ERROR.
  IF AVAILABLE Customer THEN
    DISPLAY Customer.CustNum Customer.Name Customer.PostalCode.
  ELSE LEAVE.
END.
```

ROWID

Every database record has a unique internal identifier known as the ROWID. This identifier has the data type, ROWID. You use the ROWID function to retrieve the ROWID of the database record currently in the record buffer.

The following example demonstrates using the ROWID function to retrieve the identifier of the record currently in the buffer. The identifier is later used to re-fetch the record with locking so it can be updated.

```
VAR ROWID custrid.

FIND FIRST Customer NO-LOCK.
custrid = ROWID(Customer).  // Get the rowid and save it, so it can be refetched.

IF Customer.balance > 0 THEN DO:
  FIND Customer WHERE ROWID(Customer) = custrid EXCLUSIVE-LOCK.
  Customer.Comments = "Balance remaining".   // Update the Comments field in the Customer
 record
  DISPLAY Customer.Name Customer.Balance Customer.Comments FORMAT "X(20)".
  RELEASE Customer.
END.
```

Queries

A query defines a set of data to retrieve from the database. It provides similar functionality as the data access blocks DO, FOR, and REPEAT. The main difference is that the result set defined by a query is not scoped to the block where it is defined. Using the handle to a query, you can access the query and its result set from anywhere in your application. This gives you the ability to modularize your application in ways that can't be done with block-oriented result sets.

Queries give your data access language these important characteristics:

- Scope independence — You can refer to the records in the query anywhere in your application.
- Record retrieval independence — You can move through the result set under complete control of either program logic or user events.

- Repositioning flexibility — You can position to any record in the result set at any time.

To get a query to retrieve data, you need to open it. When you are done with a query you should close it to free the system resources used by the query.

Static queries

Static queries are used when the definition of the query is known during development. You define a static query using the `DEFINE QUERY` statement to create the details for the query. The query must be opened with an `OPEN QUERY` statement before it can be used.

The following code defines and opens a query. It then cycles through all of the customer records and counts them as it goes.

```
VAR INT iCount.
DEFINE QUERY qCust for Customer.

OPEN QUERY qCust FOR EACH Customer.
GET FIRST qCust.

DO WHILE AVAILABLE Customer:
  iCount = iCount + 1.
  GET NEXT qCust.
END.
DISPLAY iCount.

CLOSE QUERY qCust.
```

Dynamic queries

Dynamic queries are used when the definition of the query is not known until runtime. For example you might want the user to input a query string. To construct a dynamic query you define a handle variable for the query and you use the CREATE QUERY statement to create an empty query at runtime. You then set the buffers using the SET-BUFFERS method and then prepare the query using the QUERY-PREPARE( ) method, where you pass the dynamic query string as a parameter. Finally, the query must be opened using the QUERY-OPEN( ) method before it can be used.

```
VAR HANDLE hQuery.

DEFINE INPUT PARAMETER bufHandle AS HANDLE.
DEFINE INPUT PARAMETER qryString AS CHARACTER.

CREATE QUERY hQuery.
hQuery:SET-BUFFERS(bufHandle).
hQuery:QUERY-PREPARE(qryString).
hQuery:QUERY-OPEN().
```

For more information, see Using Queries in *Develop ABL Applications* and Query object handle in the *ABL Reference*.

GET statement

The GET statement returns one record, and optionally related records, from an opened query.

Syntax

```
GET { FIRST | NEXT | PREV | LAST | CURRENT } query
  [ SHARE-LOCK | EXCLUSIVE-LOCK | NO-LOCK ].
```

`FIRST | NEXT | PREV | LAST | CURRENT`

- FIRST returns the first record from the query.
- NEXT returns the first or next record from the query.
- PREV returns the preceding or last record from the query.
- LAST returns the last record from the query.
- CURRENT refreshes the current record or records from the query.

`query`

The name of the query.

`SHARE-LOCK | EXCLUSIVE-LOCK | NO-LOCK`

The specified lock is applied to the record. Overrides the default locking of the OPEN QUERY statement.

Transactions

Transactions ensure that the data in your database maintains integrity. In this topic, you learn about implicit and explicit transactions, and how the `UNDO` statement undoes a transaction.

Implicit transactions

During a transaction, information on all database activity occurring during that transaction is written to a before-image (or BI) file that is associated with the database. The BI file is located on the server with the other database files. The information written to the before-image file is coordinated with the timing of the data written to the actual database files. That way, if an error occurs during the transaction, the AVM automatically uses the before-image file to restore the database to the condition it was in before the transaction started.

The statements which start an implicit transaction are:

- `FOR` blocks that directly update the database
- `REPEAT` blocks that directly update the database
- Procedure blocks that directly update the database
- `DO` blocks with the `ON ERROR` phrase that contain statements that update the database

The following example code starts a transaction for each iteration, resulting in the updates for each customer record as a separate transaction.

```
FOR EACH Customer:
  /* Customer update block */
END.
```

If your application has multiple nested blocks, each of which would be a transaction block if it stood on its own, then the outermost block is the transaction and all nested transaction blocks within it become subtransactions. All database activity occurring during a subtransaction is written to a local-before-image (or LBI) file.

A subtransaction block can be:

- A procedure block that is run from a transaction block in another procedure
- Each iteration of a FOR EACH block nested within a transaction block
- Each iteration of a REPEAT block nested within a transaction block
- Each iteration of a DO TRANSACTION or DO ON ERROR inside a transaction block

If an application error occurs during a subtransaction, all the work done since the beginning of the subtransaction is undone. You can nest subtransactions within other subtransactions. You can use the UNDO statement to programmatically undo a transaction or subtransaction.

The following example code starts a transaction at the Customer level and Order changes are done in a subtransaction:

```
FOR EACH Customer:  // Starts a transaction
  /* Customer update block */
  FOR EACH Order WHERE Order.CustNum = Customer.CustNum:  // Starts subtransaction
    /* Order update block */
  END.
END.
```

Explicit transactions

You can also start a transaction by adding the TRANSACTION keyword to a DO, FOR, or REPEAT block. If your code starts a transaction in one procedure and then calls another procedure, whether internal or external, the entire subprocedure is contained within the transaction that was started before it was called. If a subprocedure starts a transaction, then it must end within that subprocedure as well, because the beginning and end of the transaction are always the beginning and end of a particular block of code.

The following example code puts a transaction block around the whole update of both the Customer and any modified Orders:

```
DO TRANSACTION:
  DO:
    /* Customer update block */
  END.
  FOR EACH Order WHERE Order.CustNum = Customer.CustNum:
    /* Order update block */
  END.
END.  // TRANSACTION block
```

UNDO statement

A transaction is automatically undone when an unhandled error occurs that kicks you out of the transaction block. Your application logic can also undo a transaction when you detect a violation within your business logic. The UNDO statement lets you control when to cancel the effects of a transaction on your own. It also lets you define just how much of your procedure logic to undo.

You can use the UNDO keyword as its own statement. In this case, the AVM undoes the innermost containing block with the error property, which can be:

- A FOR block
- A REPEAT block
- A procedure block
- A DO block with the TRANSACTION keyword or ON ERROR phrase

The basic syntax for the UNDO statement is:

```
UNDO [ LEAVE | NEXT | RETRY | THROW error-or-stop-object-expression
     | RETURN [  ERROR | NO-APPLY ] [ return-value ] ] ]
```

In the following example code, a transaction is started at the Customer level and Order changes are done in a subtransaction:

```
FOR EACH Customer:  // Starts a transaction
  /* Customer update block */
  FOR EACH Order WHERE Order.CustNum = Customer.CustNum:  // Starts a subtransaction
    /* Order update block */
    /* If validation fails, exit */
    UNDO, LEAVE.     // This undoes the subtransaction
  END.
END.
```

You can also specify UNDO as an option on a DO, FOR, or REPEAT block. In the following example code, upon encountering an error, the current block is undone and execution resumes in the code following the block.

```
DO TRANSACTION ON ERROR UNDO, LEAVE:
  DO:
    /* Customer update block */
  END.
  FOR EACH Order WHERE Order.CustNum = Customer.CustNum:
    /* Order update block */
  END.
END.  // TRANSACTION block
```

You can also specify UNDO, LEAVE *label*, if you want to leave a specified transaction block identified by *label*, upon encountering the condition. See the LEAVE statement for more detail on using a label with LEAVE.

For more information, see Managing transactions and Understanding the UNDO concept.

Record locks

When you read records from a database table, the AVM applies a level of locking to the record that you can control, so that you can prevent conflicts where multiple users of the same data are trying to read or modify the same records at the same time. This locking does not apply to temp-tables since they are strictly local to a single ABL session and never shared between sessions. There are three locking levels:

- A NO-LOCK is a read-only option that might read incomplete transactional data.
- A SHARE-LOCK is held until the end of the transaction or the record release, whichever is later.
- An EXCLUSIVE-LOCK is held until the end of the transaction. It is then converted to a SHARE-LOCK if the record scope is larger than the transaction and the record is still active in any buffer. It is best to explicitly release the record after the update is complete.

When you read records using a FIND statement, a FOR EACH block, or the GET statement on a query, by default the record is read with a SHARE-LOCK. Another user can also read the same record using another SHARE-LOCK.

You can read records using a different lock level. If you intend to change a record, you can use the EXCLUSIVE-LOCK keyword. This marks the record as being reserved for your exclusive use. If any other user has a SHARE-LOCK on the record, an attempt to read it with an EXCLUSIVE-LOCK fails. Thus, a SHARE-LOCK assures you that while others can read the same record you have read, they cannot change it. You can also read a record using NO-LOCK.

When a transaction is undone, locks acquired within the transaction are released or they are changed to SHARE-LOCK if it locked the records prior to the transaction.

Optimistic locking

In a traditional host-based or client/server application, you can enforce what is referred to as a pessimistic locking strategy. This means that your application always obtains an EXCLUSIVE-LOCK when it first reads any record that might be updated, to make sure that no other user tries to update the same record.

In a distributed application, this technique simply does not work. If you read and pass records to a client session, your server-side session cannot easily hold locks on the records while the client is using them. When the server-side procedure ends and returns the temp-table of records to the client, the server-side record buffers are out of scope and the locks released. In addition, you would not want to maintain record locks for this extended duration, as it would lead to likely record contention.

The best way to make sure you get the locking you want is to be explicit about it. Follow these two guidelines for using locks:

- Always start a transaction before reading records, even with NO-LOCK, if you are going to update it inside the transaction.
- Release records explicitly when you are done updating them with the RELEASE statement

The following example shows first retrieving a record from the Customer table using NO-LOCK. The record is later retrieved using EXCLUSIVE-LOCK.

```
VAR ROWID custrid.

FIND FIRST Customer NO-LOCK WHERE Customer.Balance > 1000.
/* Get the rowid and save it, so it can be refetched. */
custrid = ROWID(Customer).

IF balance > 0 THEN DO TRANSACTION:
  FIND Customer WHERE ROWID(Customer) = custrid EXCLUSIVE-LOCK.
  /* Update the Comments field in the Customer record */
  Customer.Comments = "Balance remaining".
  DISPLAY Customer.Name Customer.Balance Customer.Comments FORMAT "X(20)".
  RELEASE Customer.
END.
```

For more information, see Handling Data and Locking Records in *Develop ABL Applications*.

Update records

To add records to a database, use the CREATE statement which places a newly created record in the database. All fields in the record are set to the initial values specified in the schema. The CREATE statement causes any CREATE trigger associated with the table to execute. This trigger may set fields in the record to new values.

To delete records from a database, use the DELETE statement. The DELETE statement causes any DELETE trigger associated with the table to execute.

Use the ASSIGN statement or the Assignment (=) statement to update a database record. The ASSIGN statement does not actually write records to the database until the end of a transaction, after the record goes out of scope, or after an explicit RELEASE.

The following code creates a new record in the Customer table:

```
CREATE Customer.     // Create the record
Customer.Name = "myNewCustomer".     // Assign values to fields
Customer.Address = "123 Main St.".
RELEASE Customer.     // Release the record

FIND FIRST Customer WHERE Customer.Name = "myNewCustomer".
DISPLAY Customer.Name Customer.CustNum Customer.Address.
```

For more information, see Adding and deleting records in *OpenEdge Programming Interfaces*.

Online database changes using ABL

You can make certain changes to your database schema, to ABL triggers, and to some database fields, while the database remains online, thereby ensuring the continuous operation of your ABL application. Learn more about the types of online database changes you can make in the following topics:

- Change schemas while the database is running
- Change non-structural fields of _File and _Field online
- ABL online schema changes to triggers
- Change ABL triggers online
- Change Int to Int64 online
- Rename fields using ABL
- Drop a field online

6

Temp-tables and Datasets

In the following topics, you learn about temp-tables and datasets (also known as ProDataSets). Temp-tables and datasets are important constructs for working with data in an OpenEdge database. Temp-tables are relational structures and allow you to define a set of data. Datasets are collections of one or more temp-tables and allow you to specify the relationships between tables. You can also easily convert temp-tables and datasets to XML and JSON formats for use in other applications.

For details, see the following topics:

- Temp-tables
- ProDataSets

Temp-tables

A temporary table (temp-table) is a very important ABL construct that allows you to define a set of data. Temp-tables are relational-based structures, visible to the OpenEdge session that creates them, and only last the duration of the session. Temp-tables may represent data from one or more tables in your database or they may contain different data.

Use cases

There are two main use cases for temp-tables. You can use a temp-table to represent data from one or more tables in your database. In this use case you perform similar operations on the temp-table, as you would on a database table, however the database itself is not affected. This allows operations to take place in the application logic, without any involvement from the database server. You initially fill the temp-table with data from one or more tables in the database, You can then manipulate the data in the temp-table without tying up the database. The database can be updated with changes from the temp-table at a later time, although care must be given to data integrity since the database might have changed during this time.

The second use case is when you want to work with a set of local data and even pass this set of data to another procedure or session. You can think of a temp-table in this case as a columnar structure where each row uses the same local schema definition.

Static temp-tables

Static temp-tables are used when the schema of the table is known during development. You define a static temp-table using the DEFINE TEMP-TABLE statement to create the schema for the temp-table. You create a record in the temp-table using the CREATE statement. The following example code defines a temp-table called `ttCustomer` with two fields (columns), creates a record, assigns values to the fields, and displays the record.

```
DEFINE TEMP-TABLE ttCustomer NO-UNDO     // Define the temp-table
  FIELD CustName AS CHARACTER
  FIELD CustId AS CHARACTER.

CREATE ttCustomer.     // Create a record
CustName = "John Smith".     // Assign values to the fields
CustId = "98765".

DISPLAY ttCustomer.     // Display the record
```

You can pass a static temp-table to another procedure using the `TABLE` parameter. In this case the table data is copied from one procedure to the other. Static temp-tables require a complete, static definition of the table on each side of the transfer, because the schema is not passed as part of the parameter. For more information see Using a temp-table as a parameter.

Dynamic temp-tables

Dynamic temp-tables are used when the schema of the table is not known until runtime. You use the CREATE TEMP-TABLE statement to create an empty temp-table at runtime. You then define the schema using ADD methods such as ADD-NEW-FIELD( ) and ADD-NEW-INDEX( ). Once the definition for the dynamic temp-table is complete, you call TEMP-TABLE-PREPARE( ) to signal that the table definition is complete. The temp-table is now ready to hold data.

The following example code demonstrates creating a dynamic temp-table.

```
VAR HANDLE tth.
VAR HANDLE bufferh.

/* Create an empty, undefined temp-table */
CREATE TEMP-TABLE tth.

/* Add fields to the temp-table */
tth:ADD-NEW-FIELD("custName","character").
tth:ADD-NEW-FIELD("custNum","integer").

/* Signal that the temp-table definition is complete and assign it the name "ttCust"
*/
tth:TEMP-TABLE-PREPARE("ttCust").

/* Get the buffer-handle for the temp-table */
bufferh = tth:DEFAULT-BUFFER-HANDLE.

/* Create a record and store it in the buffer */
bufferh:BUFFER-CREATE().

/* Assign values */
bufferh:BUFFER-FIELD("custName"):BUFFER-VALUE = "John Smith".
bufferh:BUFFER-FIELD("custNum"):BUFFER-VALUE = "12345".
...
```

To pass a dynamic temp-table you pass the handle itself. You can also pass a dynamic temp-table using the `TABLE-HANDLE` parameter. For more information on parameter passing, see Parameter passing syntax.

For more information on temp-tables, see Defining and Using Temp-tables and Temp-table object handle.

Include files for temp-table definitions

An ABL include file is a file that contains ABL code that is included in another procedure or class when the procedure or class is compiled. The extension of an ABL include file is `.i`. You use include files to place common code in a separate file where the common code is typically shared by other procedures or classes in your application. A best practice is to place the definitions of a temp-table that is shared by user interface logic and business logic into an include file.

The following example code shows a temp-table definition placed in a separate file called `ttOrder.i`.

```
/* ttOrder.i */

DEFINE TEMP-TABLE ttOrder NO-UNDO
  FIELD OrderNum AS INTEGER
  FIELD OrderDate AS DATE
  FIELD ShipDate AS DATE
  FIELD PromiseDate AS DATE
  FIELD OrderTotal AS DECIMAL
  INDEX OrderNum IS UNIQUE PRIMARY OrderNum.
```

To use the include file in your code, surround the pathname of the include file with curly braces (`{}`) and place it in the code where you would like it to go. A temp-table definition would typically go near the beginning. You can specify a path relative to the PROPATH environment variable.

```
/* myProcedure.p */

BLOCK-LEVEL ON ERROR UNDO, THROW.

{ttOrder.i}

/* code to populate the ttOrder temp-table*/
```

For more information, see { } Include file reference and Using include files to duplicate code.

Empty a temp-table

Static temp-tables remain for the life of the procedure where it is defined. It's a good practice to pay attention to scope of the temp-table and to empty it when it's no longer needed, or when you want a fresh set of records. Failing to empty it results in a temp-table that may contain records from the previous call, which might be unexpected and should be avoided. To empty a temp-table you use the EMPTY TEMP-TABLE statement. The basic syntax is:

```
EMPTY TEMP-TABLE temp-table-name.
```

Copy records from a database to a temp-table

You can populate a temp-table by copying records into it from the database. You can copy the entire record or select fields. To copy a record from the database table to the temp-table you must:

1. Create a temp-table record.
2. Copy the record from the database (source) into the temp-table record (target).

Syntax for creating a temp-table record

The basic syntax for creating a temp-table record is the same as for creating a database table record:

```
CREATE ttName.
```

Where *ttName* is the name of the temp-table. This creates the temp-table record.

See the CREATE statement for more detail.

Syntax for buffer-copy

Use the following syntax to copy a database record buffer into a temp-table record buffer:

```
BUFFER-COPY source TO target.
```

See the BUFFER-COPY statement for more detail.

The following example code shows how to populate the `ttCustomer` temp-table from the database. Note that the `BUFFER-COPY` only copies the matching fields; fields in the database or temp-table that do not match the other are ignored.

```
DEFINE TEMP-TABLE ttCustomer NO-UNDO     // Define the temp-table
  FIELD CustNum AS INTEGER
  FIELD Name AS CHARACTER
  FIELD City as CHARACTER
  FIELD State as CHARACTER
  FIELD Country as CHARACTER.

FOR EACH Customer WHERE Customer.Country = "USA":  // Iterate through Customer table
  CREATE ttCustomer.     // Create the temp-table record
  BUFFER-COPY Customer TO ttCustomer.  // Copy database record into temp-table record
  DISPLAY ttCustomer.
END.

/* return the temp-table to the UI client */
```

ProDataSets

A Progress DataSet, or ProDataSet, is also commonly referred to as a dataset. The ProDataSet is a very powerful construct that widens and extends the functionality of temp-tables. A ProDataSet object is basically a collection of one or more member temp-tables. It also optionally contains a collection of data relationships among the member tables.

The benefits of using ProDataSets include:

- Ability to group multiple temp-tables into a single dataset object.
- Ability to define relationships between those tables.
- Ability to associate a data source with each dataset or member temp-table.
- Ability to track changes while manipulating the dataset data.
- Ability to save the changed data back to the data source.
- Ability to pass the dataset as a single parameter from one procedure to another, within a single ABL session or between sessions.
- Simplified management of complex transactions.

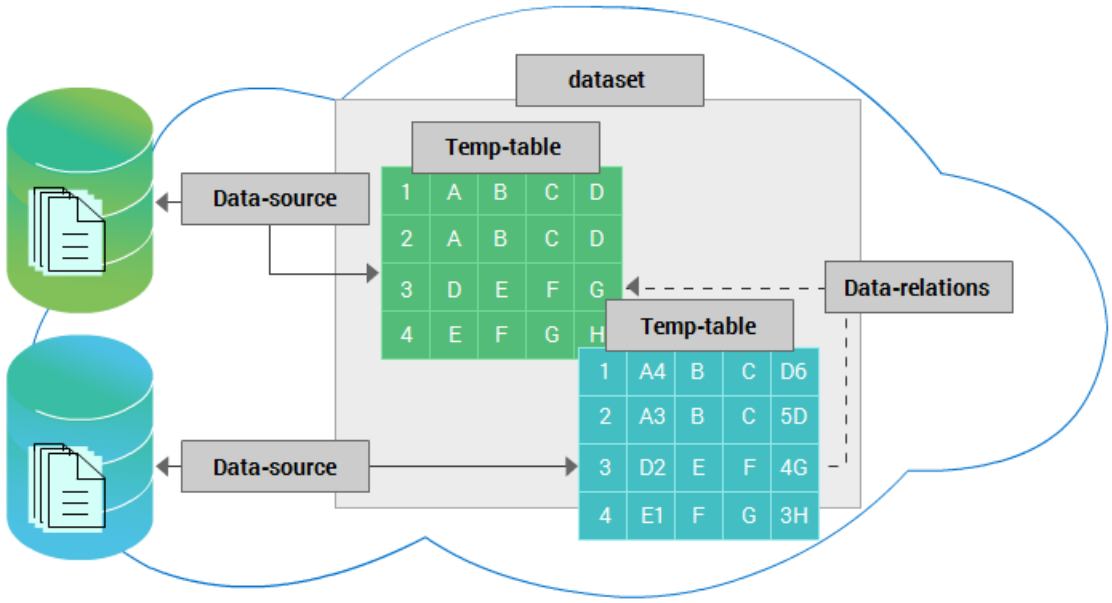
For more information, see Introduction to the OpenEdge DataSet in *Use ProDataSets*.

ProDataSet concepts

A dataset is a single, in-memory business object composed of ABL elements including temp-tables, data relationships, sources of data, and events.

A dataset often specifies the relationships between the component tables. In addition, it may be attached to data sources that can be used to populate the dataset with data. Changes to data in the dataset may then be stored back in the data sources. A dataset provides a mapping between a set of database tables or other data sources and their in-memory, possibly remote, representation.

Figure 2: ProDataSet



For more information, see ProDataSet goals and Architecture in *Use ProDataSets*

Define and use a dataset

The main steps for defining and using a dataset are:

1. Define a temp-table for each table you want to use in the dataset.
2. Define the dataset by specifying the temp-tables and their relationships (if any).
3. Identify, define, and attach the data-sources to populate the dataset.
4. Populate the dataset with data.
5. Update the dataset data and sync it back to the data-source.
6. Cleanup after using a dataset.

The first two steps are discussed next in Define a dataset on page 80. The rest of the steps are discussed in the Datasets track .

Define a dataset

This main steps for defining a dataset are:

1. Define a temp-table for each table you want to use in the dataset.
2. Define the dataset by specifying the temp-tables and their relationships (if any).

Define the temp-tables

To define a temp-table you use the DEFINE TEMP-TABLE statement. When defining a temp-table manually, you specify each field and index in the temp-table definition. You can define each field as a specific data-type, as you would in a `DEFINE VARIABLE` statement. To define a temp-table with specific fields and indexes, use this syntax:

```
DEFINE TEMP-TABLE table-name [ NO-UNDO ] BEFORE-TABLE before-table-name
     { FIELD field-name  AS data-type }
     { INDEX index-name [ IS [ UNIQUE ] [ PRIMARY ] ] { index-field } }.
```

The before table is used when a dataset is updated so that the original and updated data can be stored in the dataset before it is committed to the database.

Here is an example of the `DEFINE TEMP-TABLE` statement:

```
/* ttOrder.i */

DEFINE TEMP-TABLE ttOrder NO-UNDO BEFORE-TABLE bttOrder
  FIELD OrderNum AS INTEGER
  FIELD OrderDate AS DATE
  FIELD ShipDate AS DATE
  FIELD PromiseDate AS DATE
  FIELD OrderTotal AS DECIMAL
  INDEX OrderNum IS UNIQUE PRIMARY OrderNum.
```

Define the dataset

After you define the temp-tables, you use the DEFINE DATASET statement to define the dataset that is composed of those temp-tables. Similar to defining temp-tables in their own include files, you define a dataset in its own include file. The `DEFINE DATASET` statement is a complex statement with multiple components. You use it to:

- Name the dataset and specify the temp-tables that comprise the dataset.
- Define any data-relations between those temp-tables.

To define a dataset, you use the `DEFINE DATASET` statement. Here is the simplified syntax:

```
DEFINE DATASET dataset-name FOR temp-table-name [,temp-table-name ]...
     [ DATA-RELATION [data-relation-name] FOR parent-temp-table-name,
child-temp-table-name
     RELATION-FIELDS (parent-field1, child-field1 [, parent-fieldn, child-fieldn
 ]...) ]
```

`dataset-name`

Specifies the name of the dataset.

`temp-table-name`

Specifies the name of the temp-table(s) in the dataset.

`data-relation-name`

Specifies a data-relation object. See ProDataSet relations for more detail.

`parent-temp-table-name, child-temp-table-name`

Identifies the parent and child temp-tables for the data relation.

`parent-field`*`n`*`, child-field`*`n`*

Define the relationship between fields in the temp-tables. For the most efficient joins, the `RELATION-FIELDS` fields should be indexed.

The following example defines a dataset called `dsOrderOrderLine`. It is comprised of two temp-tables called `ttOrder` and `ttOrderLine`. It states that the relationship between the two tables is based upon the `ordernum` fields. In this example the fields in the two tables have the same name, but they don't need to be. You can also specify relation-fields that have different names.

```
/* dsOrderOrderLine.i */

{include/ttOrder.i}
{include/ttOrderLine.i}

DEFINE DATASET dsOrderOrderLine FOR ttOrder, ttOrderline
  DATA-RELATION drOrderOrderLine FOR ttOrder, ttOrderline
  RELATION-FIELDS (ordernum,ordernum).
```

You can define more than one data-relation for a dataset and include more than two temp-tables. The following is an example:

```
/* dsOrderOrderLineItem.i */
{include/ttOrder.i}
{include/ttOrderLine.i}
{include/ttItem.i}

DEFINE DATASET dsOrderOrderLineItem FOR ttOrder, ttOrderLine, ttItem
  DATA-RELATION drOrderOrderLine FOR ttOrder, ttOrderLine
    RELATION-FIELDS (Ordernum, Ordernum)
  DATA-RELATION drOrderLineItem FOR ttOrderline, ttItem
    RELATION-FIELDS (Itemnum, Itemnum).
```

You can learn more about datasets in the Datasets track or Use ProDatasets guide.