Progress®

Basic ABL

OpenEdge®

Copyright

September 2019

Last updated with new content: Release 12.1

Updated: 2019/09/30

Table of Contents

Preface.....7

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.....15

Statements.....15

Blocks.....16

Comments.....17

Variables and data types.....18

Control flow.....20

IF ... THEN ... ELSE statement.....20

CASE statement.....21

DO statement.....21

FOR statement.....23

REPEAT statement.....23

Expressions and operators.....24

Built-in functions.....25

Handle-based and class-based objects.....25

Code for portability.....27

Procedures and User-defined Functions.....29

ABL procedures.....29

User-defined functions.....34

Compile and Run.....37

R-code and the AVM.....37

COMPILE statement.....38

RUN statement.....38

Run using the command line.....38

Work with the OpenEdge Database..39
Tables and buffers..39
Data access..40
Queries..42
Transactions...44
Record locks...47
Update records..48

Temp-tables and Datasets..49
Temp-tables..49
Include files for temp-table definitions..51
Empty a temp-table..51
Copy records from a database to a temp-table...............................52
ProDataSets..52
ProDataSet concepts...53
Define and use a dataset..54

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.

See also

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 Creating a database with the PRODB utility.

Before connecting an application to a database you may have to start the database server. For more information on starting a database server see Starting up and shutting down a database server. 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.

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 on page 14

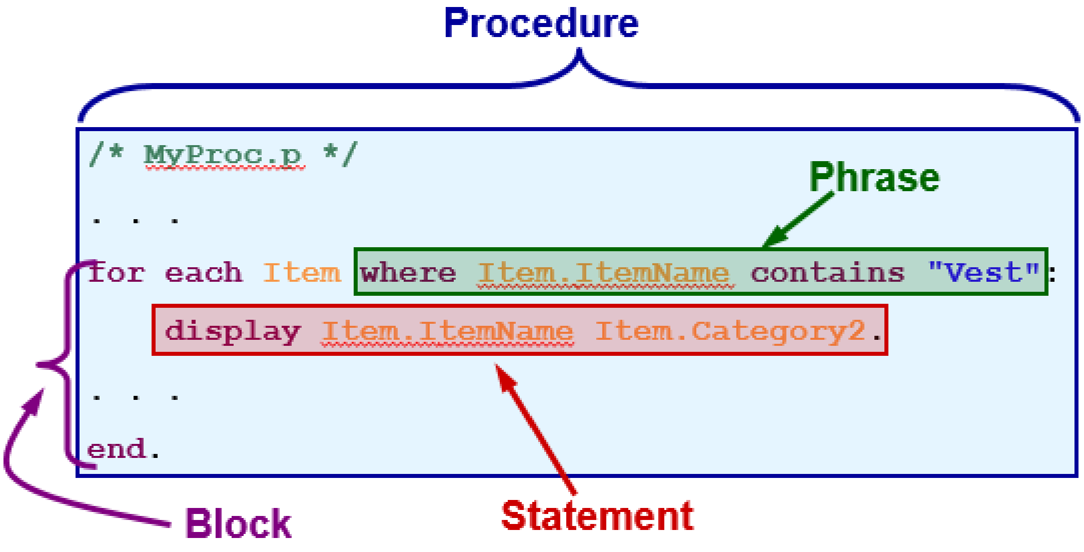
- Classes and interfaces on page 15
- Statements on page 15
- Blocks on page 16
- Comments on page 17

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



```
/* MyProc.p */
. . .
for each Item where Item.ItemName contains "Vest":
   display Item.ItemName Item.Category2.
. . .
end.
```

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.

See also

An ABL procedure consists of statements

PROCEDURE statement

FUNCTION statement

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.

See also

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.

The ABL Syntax Reference contains reference information on all the ABL statements.

See also

An ABL procedure consists of statements

ABL combines procedural, database, and user interface 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

Block name	Defined by
`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
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)

See also

Procedure Blocks and Data Access

Transactions and trigger and procedure blocks

Comments

An ABL application can also contain non-executable comments. There are two styles of comments, single-line and multi-line. 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.
```

See also

// Single-line comments

/* Multi-line comments */

Variables and data types

ABL lets you define program variables for use within an application. The variables must be defined with a specific data type. After the variables are defined you can assign values to them.

Define a variable

Variable definitions are typically placed at the beginning of your code. In ABL you do this using the DEFINE VARIABLE statement. The basic simplified syntax 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 table below for a list of supported data types.

`INITIAL` *`value`*

Optionally specifies an initial value for the variable. If specified it must be a constant. See the table below 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. Once defined you can then reference the individual array elements in your code by enclosing the array subscript in square 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.

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

Table 2: Data types

Data type name	Default initial value
CHARACTER	`""` (The empty string)
CLASS *object-type-name*	Unknown value (`?`)
DATE	Unknown value (`?`)

Data type name	Default initial value
DATETIME	Unknown value (`?`)
DATETIME-TZ	Unknown value (`?`)
DECIMAL	0
HANDLE	Unknown value (`?`)
INTEGER	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 operator (`=`). In the following example the variable, *`name`*, is set to the string, `NewCustomer`.

```
/* Set a local string variable to "NewCustomer" */

DEFINE VARIABLE name AS CHARACTER NO-UNDO.

name = "NewCustomer".
```

ABL Unknown value

For certain data types, when a value has not yet been 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; it 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 in fact the statement is asking if `ShipDate` has the Unknown value, which means it has no particular value at all. The Unknown value is like the NULL value in SQL, and the expression `IF ShipDate = ?` is like the SQL expression `IF ShipDate IS NULL`.

See also

DEFINE VARIABLE statement

Other variable qualifiers

Data types

Using program variables and data types

Assigning a value to a variable

Using the ABL Unknown value

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 on page 20
- CASE statement on page 21
- DO statement on page 21
- FOR statement on page 23
- REPEAT statement on page 23

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 example code contains an `IF THEN ELSE` statement

```
DEFINE VARIABLE custRating AS CHARACTER NO-UNDO.
DEFINE VARIABLE balance as INTEGER NO-UNDO.

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 code, the `CASE` statement is used to assign the yearly quarter based on the month.

```
DEFINE VARIABLE qtr AS CHARACTER NO-UNDO.

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

```
DEFINE VARIABLE ix AS INTEGER NO-UNDO.

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:

```
DEFINE VARIABLE ix AS INTEGER NO-UNDO.

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.
```

See also

FOR EACH CUSTOMER

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.

```
DEFINE VARIABLE ix AS INTEGER INITIAL 0 NO-UNDO.

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.

```
DEFINE VARIABLE i AS INTEGER.

i = 2.

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

Running the code produces the following output:

```
high
```

See also

Using built-In ABL functions

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.

```
DEFINE VARIABLE hSocket AS HANDLE NO-UNDO.

```

```
/* 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. See Chained handle references for more detail.

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.*.
DEFINE VARIABLE rFile AS CLASS FileInputStream NO-UNDO.
DEFINE VARIABLE mData AS MEMPTR NO-UNDO.

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

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

See also

Handle Reference

Handle Attributes and Methods Reference

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
```

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.
```

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 detail see Parameter definition syntax and Parameter passing syntax.

The following example code demonstrates passing parameters:

```
DEFINE VARIABLE v1 AS CHARACTER NO-UNDO INITIAL "XXX".
DEFINE VARIABLE v2 AS CHARACTER NO-UNDO INITIAL "YYY".
DEFINE VARIABLE v3 AS CHARACTER NO-UNDO INITIAL "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.

```
DEFINE VARIABLE hEmpLibrary AS HANDLE NO-UNDO.
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 */

DEFINE VARIABLE dtStart AS DATE NO-UNDO.
DEFINE VARIABLE dtEnd AS DATE NO-UNDO.
DEFINE VARIABLE lOK AS LOGICAL NO-UNDO.
DEFINE VARIABLE hEmpLibrary AS HANDLE NO-UNDO.

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.

See also

RETURN statement and RETURN-VALUE

Running a subprocedure

Using external and internal procedures

When to use internal and external procedures

Writing internal procedures

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):
  DEFINE VARIABLE cResult AS CHARACTER NO-UNDO.
  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.

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`.

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 parameter (`-p`). This starts a new AVM and runs the file specified by the `-p` parameter.

Basic usage

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

See also

Use the Proenv utility

Startup Procedure (-p)

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

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.

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 a database table using a database access statement, a record buffer for the table is 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.

See also

Elements of a relational database

Record buffers

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.
```

See also

Data access without looping: the FIND statement

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.

```
DEFINE VARIABLE custrid AS ROWID NO-UNDO.

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.

```
DEFINE VARIABLE iCount AS INTEGER NO-UNDO.
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.

```
DEFINE INPUT PARAMETER bufHandle AS HANDLE.
DEFINE INPUT PARAMETER qryString AS CHARACTER.
DEFINE VARIABLE hQuery AS HANDLE  NO-UNDO.

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

See also

Closing a query

Query object handle

Queries versus block-oriented data access

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`.

See also

Managing transactions

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`.

```
DEFINE VARIABLE custrid AS ROWID NO-UNDO.

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.
```

See also

Handling Data and Locking Records

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.
```

See also

Adding and deleting records

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.

```
DEFINE VARIABLE tth as HANDLE NO-UNDO.
DEFINE VARIABLE bufferh as HANDLE NO-UNDO.

/* 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 see Parameter passing syntax.

See also

Defining and Using Temp-tables

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*/
```

See also

{ } Include file reference

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.

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.
```

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 */
```

See also

BUFFER-COPY statement

CREATE statement

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.

See also

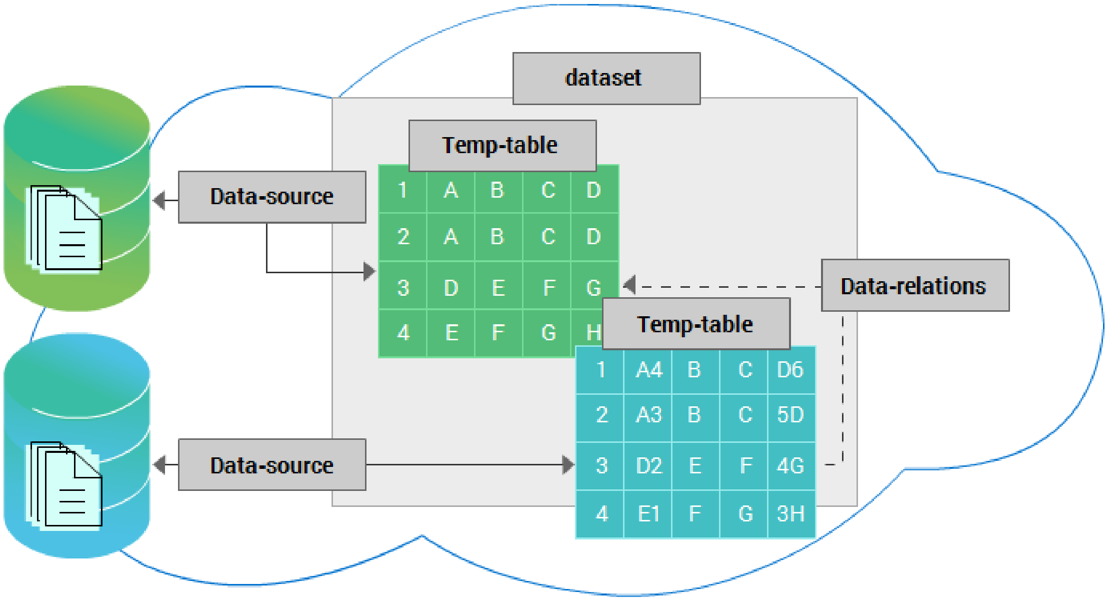
Introduction to the OpenEdge DataSet

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



See also

ProDataSet goals

Architecture

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 54. The rest of the steps are discussed in the Datasets track.

See also

Define a static ProDataSet

Populate a ProDataSet

Dynamic ProDataSet Basics

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-fieldn, child-fieldn`

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.