Progress®



OpenEdge® Web Paper:
ABL Data Types Addenda

OpenEdge®

Copyright

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

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

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

February 2019

Last updated with new content: Release 12.0

Updated: 2019/03/01

Table of Contents

Preface....7

Purpose....7
Using ABL documentation....7
References to ABL compiler and run-time features....8
References to ABL data types....8
Typographical conventions....8
Examples of syntax descriptions....9
Long syntax descriptions split across lines....11
Complex syntax descriptions with both required and optional elements....11
OpenEdge messages....12
Obtaining more information about OpenEdge messages....12

Chapter 1: Introduction....15

Data types....15
Data type terminology....16

Chapter 2: INT64 data type....17

Display Format....17
Related ABL functions....18
INT64 in arithmetic and comparison operations....18
Subtracting DATETIME values....19
Mixed INTEGER and INT64 expressions....19
Passing INT64 variables as parameters....20
Determining when to use INT64 versus INTEGER....21
Specifying not to use INT64....21

Chapter 3: Large object data types....23

Binary large objects....24
What you can do with BLOB values....24
Restrictions....24
Character large objects....25
What you can do with CLOB values....25
Restrictions....26
CLOB code pages....26
LONGCHAR data type....27
What you can do with LONGCHAR values....27
Restrictions....29

LONGCHAR code pages....29

Chapter 4: Using large objects in ABL....31

Copying large objects....31
Converting code pages for CLOB and LONGCHAR values....32
Deleting large objects....33
Dumping and loading large objects....33
Dumping large objects....33
Dumping all data except large objects....34
Loading large objects....35
UNDOing subtransactions that contain large object fields....35
Passing LONGCHAR variables as parameters....36
Modifying BLOB data....36
Using BLOB values and image files....37
Using BLOB values and word processor files....37
Creating a CLOB from a file....38

Chapter 5: DATETIME and DATETIME-TZ data types....39

DATETIME data type....40
DATETIME-TZ data type....40
Summary of features....40
Related ABL functions....41
SESSION handle attributes....42
Precedence of default time zone sources for ABL time date functions....43
DATETIME and DATETIME-TZ literals....43
Comparing date and datetime values....44
Display formats....44
Displaying DATETIME data based on display format....45
Displaying DATETIME-TZ data based on display format....45
DATETIME and DATETIME-TZ limits....46
Datetime arithmetic....46
Adding and subtracting basic datetime values....46
Adding and subtracting days plus milliseconds with a datetime value....47
Adding, subtracting, and obtaining specific time intervals....48
Passing DATETIME and DATETIME-TZ as parameters....49
Dumping and loading DATETIME and DATETIME-TZ fields....50

Chapter 6: Arrays....51

Arrays as parameters....51
Passing indeterminate arrays as parameters....52
Deep copying arrays....54

Preface

For details, see the following topics:

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

Purpose

This web paper documents new data type support which applies to OpenEdge® Releases 10.0B and later releases.

Using ABL documentation

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

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

References to ABL compiler and run-time features

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

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

References to ABL data types

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

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

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

Typographical conventions

This documentation uses the following typographical and syntax conventions:

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

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

Examples of syntax descriptions

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

Syntax

```
ACCUM aggregate expression
```

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

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

Long syntax descriptions split across lines

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

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

Syntax

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

Complex syntax descriptions with both required and optional elements

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

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

Syntax

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

OpenEdge messages

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

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

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

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

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

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

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

Obtaining more information about OpenEdge messages

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

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

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

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

1. Start the Procedure Editor:

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

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

1

Introduction

For details, see the following topics:

- Data types
- Data type terminology

Data types

The *data type* of a database field, temp-table field, or program variable defines what kind of data the field or variable can store. To allow better interoperability between OpenEdge® applications and external products, OpenEdge provides additional data types and new features for some existing data types in ABL (Advanced Business Language).

Note: For a list of all the ABL data types, see *OpenEdge Development: ABL Reference*.

The new data types and the ability to pass arrays as parameters provides a higher level of interoperability with products such as SQL92, XML, Open Client, Application Server, and Web services products. When the interface of an application is defined by a non-ABL application, these new data types ensure that the application executes properly. In addition, these new data types allow ABL developers to create more data-robust OpenEdge applications.

This new data type support applies to OpenEdge Releases 10.0B and later releases. It is described in the following sections:

- Data type terminology on page 16

- INT64 data type on page 17
- Large object data types on page 23
- Using large objects in ABL on page 31
- DATETIME and DATETIME-TZ data types on page 39
- Arrays as parameters on page 51
- Deep copying arrays on page 54

Data type terminology

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

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

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

2

INT64 data type

Starting with OpenEdge®Release 10.1B, ABL includes a 64-bit integer data type, `INT64`. This data type is separate from the existing 32-bit data type (`INTEGER`). The `INT64` data type is compatible with all current functions and statements that currently support the 32-bit integer. That is, `INT64` and `INTEGER` are interchangeable everywhere in ABL, except in parameters and return types for strongly-typed class-based methods where you want the type checking and where you control both the caller and callee routines.

For details, see the following topics:

- Display Format
- Related ABL functions
- INT64 in arithmetic and comparison operations
- Passing INT64 variables as parameters
- Determining when to use INT64 versus INTEGER
- Specifying not to use INT64

Display Format

The default display format for `INT64` data type is "->,>>>,>>9" (the same as the default display format for `INTEGER`.) You can define an `INT64` field or variable to be viewed as (`VIEW-AS`) a fill-in field, text widget, combo-box, radio-set, or slider.

Note that if you specify `VIEW-AS SLIDER` for an `INT64` field or variable, the value must still remain within the -2147483648 to +2147483647 integer range.

Related ABL functions

To support the INT64 data type, ABL supports the following new functions and statements:

- **INT64 function** — Takes any data type and returns an INT64 value, if the conversion is possible. This function takes most common data types except for RAW, MEMPTR, or LONGCHAR.
- **PUT-INT64 statement** — Stores the signed 64-bit value of an INT64 or INTEGER expression at the specified memory location. The specified memory location can have a RAW or a MEMPTR value.
- **GET-INT64 function** — Returns the signed 64-bit value at the specified memory location as an INT64 value. The specified memory location can have a RAW or a MEMPTR value.
- **PUT-UNSIGNED-LONG statement** — Stores an INTEGER or INT64 value as a 32-bit unsigned value to a MEMPTR or RAW value. This statement is analogous to PUT-UNSIGNED-SHORT, except with a 32-bit value.
- **GET-UNSIGNED-LONG function** — Returns the unsigned 32-bit value at the specified memory location as an INT64. This is analogous to GET-UNSIGNED-SHORT, except with a 32-bit value.

In addition, the following functions, which were compiled as returning INTEGER in Release 10.1A and earlier, are now compiled as returning INT64:

- CURRENT-VALUE
- DYNAMIC-CURRENT-VALUE
- DYNAMIC-NEXT-VALUE
- ETIME
- GET-POINTER-VALUE
- GET-SIZE
- INTERVAL
- NEXT-VALUE
- SEEK

You can override this result and have these functions compile to return INTEGER with either the INTEGER function or by using the No INT64 (-noint64) startup parameter. For more information on this startup parameter, see Specifying not to use INT64 on page 21.

INT64 in arithmetic and comparison operations

Arithmetic operations are unchanged by the new INT64 data type. That is, operators (such as the * Multiplication and the + Addition operators) are still compiled as INTEGER if both arguments are INTEGER.

Starting with Release 10.1B, all intermediate calculations are carried out in 64-bit arithmetic. With the introduction of the INT64 data type, any operator or function that used to return an INTEGER can now return a value greater than 32-bits. This large value is held correctly as an intermediate value and does not generate an error message **unless** you assign the final result into an INTEGER database field, temp-table field, program variable, or parameter. This means that if you try to store a value outside the -2147483648 to +2147483647 range in an INTEGER, you get an error message. For example, 2,000,000,000 * 100 / 100 gives the correct result whether the target field is INTEGER or INT64. However, although 2,000,000,000 * 100 does not cause an overflow, the result must be assigned to an INT64 field. It generates a run-time error if you assign it to an INTEGER field.

If you have an existing field, program variable, or parameter defined with the INTEGER data type, and you want to make it 64-bit capable, you must change the data type to INT64. You can change the data type of a database field via the Data Dictionary tool. Note that you can make this change without having to dump and load the data. However, when you change the data type of a database field, you change the Cyclic Redundancy Check (CRC) of the database table that contains the field. Therefore, you must recompile your r-code. In addition, you should review your code for any references to the database field to determine whether there is a possibility of an overflow with the INTEGER data type.

Subtracting DATETIME values

Starting with OpenEdge Release 10.1B, you can subtract two DATETIME values, with the result being the INT64 number of milliseconds. For more information on subtracting DATETIME values, see Datetime arithmetic on page 46.

Mixed INTEGER and INT64 expressions

If arithmetic or comparison expressions contain a mix of INTEGER and INT64 data types, the result is an INT64. Dividing an INT64 with another INT64 produces a DECIMAL result, as does dividing two INTEGERs.

INT64 support applies to all of the ABL built-in methods and functions that take integer-expression parameters. That is, integer-expression parameters can be either INT64 expressions or INTEGER expressions.

Note that you **cannot** pass an INT64 expression into a class-based method that is expecting an INTEGER. Instead, you can use the INTEGER function to convert the expression to INTEGER, which the method is expecting. In particular, you must convert the sequence functions (NEXT-VALUE, CURRENT-VALUE, DYNAMIC-CURRENT-VALUE, DYNAMIC-NEXT-VALUE) to INTEGER if you want to pass them to a class method expecting INTEGER.

The following code example demonstrates arithmetic operations using a mix of INTEGER and INT64 expressions:

```
DEF VAR x AS INTEGER.
DEF VAR y AS INTEGER.
DEF VAR z AS INTEGER.
DEF VAR big AS INT64.
/* NOTE: as far as the compiler is concerned, x + y is an INTEGER*/
/* NOTE: all failures are RUNTIME failures, not COMPILER failures*/

x = 2000000000.        /*should succeed, just under 2gig*/
y = 2000000000.        /*should succeed, just under 2gig*/

 DISPLAY x + y FORMAT ">>>,>>>,>>>,>>>" .
/*should succeed if format is big enough*/
z = x + y              /* assignment >2gig to INTEGER should fail*/

INTEGER( x +           /*INTEGER function on >2gig should fail*/
big = x + y.           /*assignment >2gig to INT64 should succeed*/

DISPLAY x + y > 2000000000.
                       /*compare >2gig with INTEGER constant should succeed*/

DISPLAY x + y > 3000000000.
                       /*compare >2gig with INT64 constant should succeed */

PROCEDURE myproc:
  DEF INPUT PARAMETER mp AS INTEGER.
  DISPLAY mp.
END.

RUN myproc(big).       /*INT64 with >2gig should fail*/
RUN myproc(x + y)      /*INTEGER expression with >2gig should fail*/
big = 3.
RUN myproc(big).       /*INT64 with <2gig should succeed*/

big + x.               /*is compiled as returning INT64*/
x + y.                 /*is compiled as returning INTEGER*/

y = NEXT-VALUE(next-cust-num).          /*sequence is compiled with INT64
datatype if the sequence is up to >2gig, this will fail*/

big = NEXT-VALUE(next-cust-num).       /*sequence is compiled with INT64
datatype if the sequence is up to >2gig, this will still succeed*/
```

Passing INT64 variables as parameters

When using INT64 and INTEGER parameter combinations between an ABL client or an Open Client and an AppServer, the same rules apply as for local parameters. If the parameter is a ProDataSet or a temp-table, then the data types of the fields in the tables must match exactly. That is, a temp-table with an INT64 field does not match a parameter on the other side with an INTEGER field in the same position, and vice versa.

However, for simple parameters you can have an INT64 on one side and an INTEGER on the other side. If the parameter is an INPUT INTEGER parameter in the callee, it is checked for overflow in the callee. Likewise, if the parameter is an OUTPUT INTEGER, or an OUTPUT INT64, and the caller's type is INTEGER, it is checked for overflow in the caller and generates an error during callee output processing. This means that neither the ABL client nor the Open Client need to do any checking, since the callee (the AppServer) makes sure no violations can happen on both input and output. However, the client might need to use INT64 so that there are no overflows in the proxies.

Caution: If you specify NO-SCHEMA when you define a temp-table, the INT64/INTEGER parameter matching is not checked, just as other types of checking are bypassed when you specify NO-SCHEMA.

Determining when to use INT64 versus INTEGER

Even though the INT64 data type is available, Progress Software Corporation does not recommend that you completely change all INTEGER expressions to INT64 expressions in your existing applications, nor that you use INT64 exclusively in new applications. External interfaces, such as .NET, ODBC, JAVA, and OCX, will not work smoothly or efficiently if you do. There is no performance penalty for using INT64 in ABL (as all values are stored internally as INT64 and all operations are performed as INT64), but do not ignore your application's interactions with external interfaces.

Instead of a global change, you should plan carefully and specify as INT64 only those fields, program variables, and so forth, that actually have a chance of needing a value outside the -2147483648 to +2147483647 range of INTEGER data.

In deciding whether a field will need more than 2^31-1, consider not only how large the field needs to be at the present time, but also consider how large the field will need to be over the life span of the application. As an example, consider the CustNum field in the sample Sports database. After being in business a long time, the customer numbers of old (now non-existent) customers usually cannot be re-used, so they occupy CustNum values even though the total of current customers is far less.

Specifying not to use INT64

OpenEdge provides a new No INT64 (-noint64) startup parameter in case you have an unusual circumstance in which you want to execute 10.1B r-code with 10.1A executable. If you specify the -noint64 startup parameter:

- The functions that are compiled as INT64 (CURRENT-VALUE, DYNAMIC-CURRENT-VALUE, DYNAMIC-NEXT-VALUE, ETIME, GET-POINTER-VALUE, GET-SIZE, INTERVAL, NEXT-VALUE, and SEEK) are compiled as INTEGER.
- INT64-size constant values are compiled as DECIMAL
- ABL source code references to INT64 variables and fields generate errors.
- Code that subtracts two DATETIME values generates a compile-time error
- The XML schema long data type (64-bit signed integer) maps to the ABL DECIMAL data type rather than the INT64 data type. This mapping applies to:
  - The bprowsdldoc and bproxsdto4gl command line utilities
  - The Web services Client run-time product

Note: The -noint64 startup parameter does not affect performance. The arithmetic operations in Release 10.1A and earlier code are calculated using the INTEGER data type while those in Release 10.1B code are calculated using the INT64 data type.

3

Large object data types

Before OpenEdge® Release 10, all data types, other than MEMPTR, were limited in size to 32K. Release 10 introduces three new large object data types that expand this size limit:

- **Binary large object (BLOB)** — For database and temp-table fields. This data type allows the storage of large binary data in the database. Data previously stored external to the OpenEdge database can now be integrated into the OpenEdge database and accessed by ABL. This data type is the database counterpart of the MEMPTR data type.
- **Character large object (CLOB)** — For database and temp-table fields. This data type allows the storage of large character strings in the database. It also permits automatic code page translation and null-termination. CLOB values can be in any code page that is supported by the ABL code page conversion map (convmap.p) file. The code page information is retained by the CLOB. ABL ensures that all characters are valid with respect to the code page of the CLOB. This data type is the database counterpart of the LONGHAR data type.
- **LONGCHAR data type** — For in-memory use. This data type is the program variable counterpart of the CLOB data type. LONGCHAR allows much of the ABL character handling capabilities to be applied to large strings. 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.

BLOB and CLOB fields can be up to 1GB in size. LONGCHAR (and MEMPTR) variables can be any size, but they are limited by system resources.

For details, see the following topics:

- Binary large objects
- Character large objects
- LONGCHAR data type

Binary large objects

A binary large object (BLOB) contains unstructured data and consists of a group of bytes. ABL has no special knowledge about its contents. Rather, ABL assumes that the data in a BLOB is managed by other software (such as a word processor, a spreadsheet program, or a bar-code reader). When working with BLOB values, offsets are always in bytes. With BLOB values, you can store data that is up to 1GB in size. For example, if you have an application with a catalogue of images you can now easily store those images in an OpenEdge database.

A BLOB field is a database table or temp-table field that contains a BLOB locator, which points to the associated BLOB data stored in the database. You cannot manipulate BLOB data directly. Instead, you must copy a BLOB to a MEMPTR, and then use the MEMPTR functionality to manipulate the binary contents of a BLOB field in ABL. (A MEMPTR contains a sequence of bytes in memory. You can think of the MEMPTR as a cache for the BLOB.) To copy data between a BLOB field and a MEMPTR, you can use either the COPY-LOB statement or you can use an assignment. See Copying large objects on page 31 for more information.

What you can do with BLOB values

You can do the following with BLOB values:

- You can define a database table field or a temp-table field as a BLOB.
- You can use the COPY-LOB statement to copy large object data from one BLOB or MEMPTR to another. COPY-LOB also handles moving BLOB or MEMPTR data to or from the file system. For more information, see Copying large objects on page 31.
- You can EXPORT and IMPORT BLOB data to and from disk.
- You can use the LENGTH() function on a BLOB field to get the size of the BLOB in bytes.
- The BUFFER-COPY and BUFFER-COMPARE statements operate on BLOB fields. A BUFFER-COPY of a record with a BLOB in it results in the creation of a BLOB for the new record. BUFFER-COMPARE compares the data in the BLOB fields and reports any discrepancy. To skip BLOB fields during BUFFER-COPY and BUFFER-COMPARE, use the NO-LOBS option.
- You can use the dynamic BUFFER-FIELD:BUFFER-VALUE for a BLOB field in the same places you can use a static reference.
- To determine whether the buffer has large object fields defined in it, you can use the HAS-LOBS attribute for a BUFFER object. You can then check the data type of the buffer fields for the BLOB data type.

For examples on how to use BLOB fields in an application, see Using large objects in ABL on page 31.

Restrictions

The following restrictions apply to BLOB fields:

- The size limit for a BLOB is 1 gigabyte.
- You cannot use BLOB values in an index.
- You cannot have an array of BLOB fields.

- You cannot pass a BLOB as a parameter (but you can pass a temp-table that contains a BLOB field as a parameter).
- BLOB fields are only allowed in temp-tables that are NO-UNDO.
- You cannot define a local program variable as a BLOB.
- You cannot use ASSIGN triggers for a BLOB field.
- The initial value for a BLOB must be the Unknown value (?).
- You must copy a BLOB to a MEMPTR to read or edit it.
- The RAW-TRANSFER statement skips BLOB fields, assigning the Unknown value (?) to the BLOB field in the target record.

Character large objects

A character large object (CLOB) is distinguished from a BLOB in that it contains only character data, has a code page associated with it, and when working with CLOB values offsets are always in terms of characters (since some characters might be multi-byte).

Similar to a BLOB field, a CLOB field is a database table or temp-table field that contains a CLOB locator, 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 display a CLOB field by copying it to a LONGCHAR, and displaying the LONGCHAR in a large editor. See Copying large objects on page 31 for more information.

What you can do with CLOB values

You can do the following with CLOB values:

- You can define a database table field or a temp-table field as a CLOB.
- When you define a field as a CLOB, you can specify a code page for it.
- You can use the COPY-LOB statement to copy large object data from one CLOB or LONGCHAR to another. COPY-LOB also handles moving CLOB or LONGHAR data to or from the file system. For more information, see Copying large objects on page 1.
- You can EXPORT and IMPORT CLOB data to and from disk.
- The BUFFER-COPY statement operates on CLOB fields. A BUFFER-COPY of a record with a CLOB in it results in the creation of a CLOB for the new record. Use the NO-LOBS option if you want to skip CLOB fields during BUFFER-COPY.
- You can use the dynamic BUFFER-FIELD:BUFFER-VALUE for a CLOB field in most of the same places you can use a static reference.
- To determine whether the buffer has large object fields defined in it, you can use the HAS-LOBS attribute for a BUFFER object. You can then check the data type of the buffer fields for the CLOB data type.

For examples on how to use CLOB fields in an application, see Using large objects in ABL on page 31.

Restrictions

The following restrictions apply to CLOB fields:

- The UNDEFINED code page is not allowed for CLOB values.
- You must copy a CLOB to a LONGCHAR in order to read or edit it.
- The size limit for a CLOB is 1 gigabyte.
- You cannot use CLOB values in an index.
- You cannot have an array of CLOB fields.
- You cannot pass a CLOB as a parameter (but you can pass a temp-table that contains a CLOB field as a parameter).
- CLOB fields are only allowed in temp-tables that are NO-UNDO.
- You cannot define a local program variable as a CLOB.
- You cannot use ASSIGN triggers for a CLOB field.
- You cannot use BUFFER-COMPARE on a CLOB field.
- The initial value for a CLOB must be the Unknown value (?).
- The RAW-TRANSFER statement skips CLOB fields, assigning the Unknown value (?) to the CLOB field in the target record.

CLOB code pages

There are two types of CLOB values:

- **DBCODEPAGE (for database tables) or TTCODEPAGE (for temp-tables)** — A DBCODEPAGE CLOB is a field in a database that is in the database's codepage and uses the collation table associated with the database's codepage. A TTCODEPAGE is a CLOB field in a temp-table that is in the code page set by the Internal Code Page (-cpinternal) startup parameter. Also, if you define a temp-table field to be LIKE a DBCODEPAGE database field, the -cpinternal code page is used for the temp-table field.
- **COLUMN-CODEPAGE** — When you define a CLOB field (either in a database or a temp-table) you can use COLUMN-CODEPAGE to specify the code page of the CLOB as well as a valid collation for that code page.

If you do not specify a code page for a CLOB field in a temporary table, the default code page is -cpinternal. In the following example, clopcp and clopdb are CLOB fields defined elsewhere. The clopcp field is a COLUMN-CODEPAGE type of CLOB with code page 1252. The clopdb field is a DBCODEPAGE type of CLOB with code page UTF-8. The code page set by -cpinternal is iso8859-1:

```
DEFINE TEMP-TABLE ttab
   FIELD ttab1 AS CLOB
   FIELD ttab2 AS CLOB COLUMN-CODEPAGE "UTF-16"
   FIELD ttab3 LIKE clobcp
   FIELD ttab4 LIKE clobdb
   FIELD ttab5 LIKE clobcp TTCODEPAGE.
```

In this example, ttab1 is in -cpinternal (iso8859-1) because that is the default code page for CLOB fields in a temp-table. The code page for ttab2 is UTF-16 because it is specifically set using the COLUMN-CODEPAGE option. The ttab3 field has the 1252 code page because it is defined as being LIKE clobcp which defines the code page of that CLOB as 1252.

The ttab4 field has the -cpinternal code page (iso8859-1) since it is defined LIKE clobdb which is a DBCODEPAGE CLOB. It is therefore converted to the code page of the temp-table, which is always defined by -cpinternal. (You can use the DBCODEPAGE function to get the name of a connected database's code page. For more information about the DBCODEPAGE function, see *OpenEdge Development: ABL Reference*.)

The ttab5 field is in TTCODEPAGE (-cpinternal). It is defined to be LIKE clobcp but the TTCODEPAGE keyword is specified and that overrides the code page of clobcp.

LONGCHAR data type

The LONGCHAR data type consists of character data that is not limited in size to 32K, but rather it is limited in size only by the system resources. With the LONGCHAR data type, you can apply much of the existing ABL character handling capabilities to strings larger than 32K bytes.

LONGCHAR is the program variable counterpart of the CLOB data type. The LONGCHAR data type has metadata associated with it to designate its code page and other information. The code page associated with LONGCHAR data defaults to -cpinternal. However, you can use the FIX-CODEPAGE statement, CODEPAGE-CONVERT function, and the COPY-LOB statement to create a LONGCHAR in any code page supported by the convmap.cp file. To use most of the character functions available in ABL, the LONGCHAR is converted to -cpinternal.

What you can do with LONGCHAR values

You can do the following with LONGCHAR values:

- Display and update LONGCHAR in a large editor. The default is no display.
- Use LONGCHAR variables as parameters and return values in procedures, user-defined functions, and methods. Note that, as is also the case with MEMPTR values, LONGCHAR return values from user-defined functions are not allowed to be passed from an AppServer to a client.
- Use the COPY-LOB statement to copy data from a CLOB to a LONGCHAR so that you can manipulate the CLOB data.
- The following tables list the ABL elements that work with LONGCHAR variables. For example, you can get the character length of a LONGCHAR by using LENGTH with the character (default) option. Or you can SUBSTRING a LONGCHAR with the offset given in characters, not bytes. The particulars of which string manipulation functions apply to LONGCHAR values vary for each function and for particular characteristics of the LONGCHAR. For example, most functions which alter the string only apply to LONGCHAR values that are in -cpinternal. All functions which deal with LONGCHAR values and have offset and length input parameters use character offsets and lengths, not byte values. For more information on these ABL elements, see *OpenEdge Development: ABL Reference*.

Table 1: ABL operators that work with LONGCHAR variables

Operators		
Concatenation (+)	Equal (EQ or =)	Less than (LT or <)

Operators		
Assignment (=)	Greater than (GT or >)	Less than or equal (LE or >=)
BEGINS	Greater than or equal (GT or >=)	
MATCHES	Not equal (NE or <>)	

Table 2: ABL functions that work with LONGCHAR variables

Functions		
BASE64-DECODE	INDEX	R-INDEX
BASE64-ENCODE	IS-CODEPAGE-FIXED	REPLACE
CAPS	LC	RIGHT-TRIM
CODEPAGE-CONVERT	LEFT-TRIM	SHA1-DIGEST
COMPARE	LENGTH	SUBSTITUTE
ENCRYPT	LOOKUP	SUBSTRING
ENTRY	MD5-DIGEST	TRIM
GENERATE-PBE-KEY	NORMALIZE	
GET-CODEPAGE	NUM-ENTRIES	

Table 3: ABL statements that work with LONGCHAR variables

Statements		
ASSIGN	EXPORT	OVERLAY
COPY-LOB	FIX-CODEPAGE	PUT-STRING
DEFINE PARAMETER	FUNCTION	SUBSTRING
DEFINE VARIABLE	IMPORT	
ENTRY	METHOD	

Table 4: ABL attributes that work with LONGCHAR variables

Attributes	
FRAGMENT	SYMMETRIC-ENCRYPTION-KEY

Table 5: ABL methods that work with LONGCHAR variables

Methods		
DECLARE-NAMESPACE()	READ-XMLSCHEMA()	WRITE-DATA-ELEMENT()
END-ELEMENT()	SAVE()	WRITE-EMPTY-ELEMENT()
GET-CGI-LONG-VALUE()	SET-INPUT-SOURCE()	WRITE-ENTITY-REF()
GET-NODE()	SET-OUTPUT-DESTINATION()	WRITE-EXTERNAL-DTD()
GET-SERIALIZED()	SET-PARAMETER()	WRITE-FRAGMENT()
INSERT-ATTRIBUTE()	SET-SERIALIZED()	WRITE-PROCESSING-INSTRUCTION()
LOAD()	START-ELEMENT()	WRITE-XML()
LONCHAR-TO-NODE-VALUE()	WRITE-CDATA()	WRITE-XML-SCHEMA()
NODE-VALUE-TO-LONGCHAR()	WRITE-CHARACTERS()	
READ-XML()	WRITE-COMMENT()	

Restrictions

The following restrictions apply to LONGCHAR values:

- Whatever the code page, the LONGCHAR can only contain **one** string in **one** code page. It contains only 1 NULL terminator of its own code page. LONGCHAR cannot consist of multiple code pages, with different character sets in different frames, as an HTML file can.
- The UNDEFINED code page is not allowed for LONGCHAR values.
- You can use only the large editor for viewing a LONGCHAR (VIEW-AS EDITOR LARGE).
- A LONGCHAR program variable cannot be UNDO.
- You cannot use a LONGCHAR program variable in WHERE clauses.

LONGCHAR code pages

The default value of the code page for a LONGCHAR is the -cpinternal code page. To set the code page of a LONGCHAR variable to a different code page, use the following FIX-CODEPAGE statement:

Syntax

```
FIX-CODEPAGE ( longchar ) = codepage
```

You can only use this statement to set the code page for a `LONGCHAR` variable that is empty. If you try to use the `FIX-CODEPAGE` statement for a `LONGCHAR` variable that already has a value, the ABL Virtual Machine (AVM) generates an error.

When set to a valid code page, the code page of the specified variable is fixed and overrides any default behavior in assignment operations. For assignment rules, see *OpenEdge Development: ABL Reference*.

4

Using large objects in ABL

This section provides details and shows examples for using large objects in ABL.

For details, see the following topics:

- Copying large objects
- Deleting large objects
- Dumping and loading large objects
- UNDOing subtransactions that contain large object fields
- Passing LONGCHAR variables as parameters
- Modifying BLOB data
- Using BLOB values and image files
- Using BLOB values and word processor files
- Creating a CLOB from a file

Copying large objects

The `COPY-LOB` statement copies large object data from one `BLOB` or `MEMPTR` (or one `CLOB` or `LONGCHAR`) to another. It also handles copying data to or from the file system. The `CONVERT` phrase allows code page conversion of `CLOB` values and `LONGCHAR` values.

Here is the syntax for the `COPY-LOB` statement:

Syntax

```
COPY-LOB
  [ FROM ] { [ OBJECT ] source-lob | FILE source-filename }
     [ STARTING AT n ] [ FOR length ]
  TO { [ OBJECT ] target-lob [ OVERLAY AT n [ TRIM ] ] |
     FILE target-filename [ APPEND ] }
  [ NO-CONVERT | CONVERT convert-phrase ]
  [ NO-ERROR ] .
```

You can copy the entire contents of the source to the target with COPY-LOB. You can also use the COPY-LOB statement to copy part of a file (for example, using STARTING AT or FOR) to the target. For more information about the COPY-LOB statement, see *OpenEdge Development: ABL Reference*

You can also use the ASSIGN statement to copy large object data between the database, temp-table fields, and memory. For example, the following ASSIGN and COPY-LOB statements (where m2 is a MEMPTR) are functionally equivalent:

```
ASSIGN employee.blob = m2.
COPY-LOB FROM m2 TO employee.blob.
```

You can assign large object data from one BLOB or MEMPTR to another, and from one CLOB or LONGCHAR to another. You cannot assign large object data between BLOB and CLOB values or MEMPTR and LONGCHAR values. However, you can accomplish this indirectly by using the COPY-LOB statement.

Converting code pages for CLOB and LONGCHAR values

The NO CONVERT option for the COPY-LOB statement means no code page conversion is done. However, if the target is a LONGCHAR or CLOB, validation occurs for the presumed target code page.

Alternately, the CONVERT option for the COPY-LOB statement has a *convert-phrase* that lets you specify how the AVM converts object data:

Syntax

```
{
  [ SOURCE CODEPAGE codepage ]
  [ TARGET CODEPAGE codepage ]
}
```

Specify SOURCE CODEPAGE to indicate that a source object is in the specified code page. If you specify TARGET CODEPAGE, the AVM converts the target object to the specified code page.

For details on the default character conversions the AVM performs when copying data between the source and target objects, see the COPY-LOB statement entry in *OpenEdge Development: ABL Reference*.

Deleting large objects

To delete a large object, use the ASSIGN statement to set the value of the large object field to the Unknown value (?). For example:

```
ASSIGN employee.blob = ?.
```

Note that when you delete a record in a table, any non-null BLOB fields referenced in the record are also deleted.

Dumping and loading large objects

Use the EXPORT and IMPORT statements to copy BLOB and CLOB data to and from disk.

Note: You can also use the Data Administration tool to IMPORT and EXPORT large objects and to specify a directory for large object files.

Dumping large objects

When you use EXPORT to dump large object (LOB) data from a database, a uniquely named file with a .blb extension is generated for each BLOB or CLOB. By default, the .blb file is written to the same location as the file in the OUTPUT TO statement. You can use the optional LOB-DIR *dir-name* phrase on the OUTPUT TO statement, where *dir-name* evaluates to the name of the directory to write the .blb file to.

EXPORT stores the file names in the exported file (typically a .d file), in the position where the ordinary field data would be placed if the field contained non-LOB data. If the LOB field has the Unknown value (?), the AVM writes a question mark (?) in that place; if it is a zero-length LOB, the AVM creates a file as it does for any BLOB field, but the file has a zero length.

In the following example, the contents of the BLOB field is **not** written to the blob.txt file. Instead, the contents of the BLOB field is written to a new file with a .blb extension (for example, blob-field_123456.blb) and only the name of the file containing the BLOB data is written to blob.txt:

```
OUTPUT TO blob.txt.
EXPORT table.blob-field.
OUTPUT CLOSE.
```

Note that if the OUTPUT TO statement in this example included the LOB-DIR*dir-name* option, then the blob-field_123456.blb file would be created in the LOB-DIR directory. However, the AVM does **not** create the LOB-DIR directory if it does not already exist.

In the following example, the names of the files that contain the BLOB data are written to the customer.d file in the C:\DumpedData\ directory and the files are written to C:\DumpedData\custblob:

```
OUTPUT TO C:\DumpedData\customer.d LOB-DIR "custblob".

FOR EACH customer:
EXPORT customer.
END.

OUTPUT CLOSE.
```

Here is an excerpt from the contents of the customer.d file:

```
"lift line skiing", 1, "photo_123456.blb", 617-9666673, "boston"
"off the wall",     2, "photo_234567.blb", 781-5555554, "new york"
"flying fat",       3, ?                 , 303-9992222, "washington"
```

Note that the third customer had no data in the photo field and therefore is the Unknown value (?). Note also that the generated filename is always unique. If the LOB-DIR phrase is left off of the OUTPUT TO statement, the BLOB files are created in the C:\DumpedData directory.

For CLOB values, the code page of the CLOB is part of the filename generated by the EXPORT statement. The file name consists of the field or variable name, followed by the code page name surrounded by exclamation marks (!), followed by the unique name and the .blb extension. For example:

```
Fclob!UTF-8!12345_6.blb
```

Dumping all data except large objects

You can specify to dump all data, except for large objects with the NO-LOBS option on the EXPORT statement. Here is an example:

```
FOR EACH customer:
 EXPORT customer NO-LOBS.
END.
```

Here is the output showing that the BLOB data was excluded from the data dump and is instead represented by the Unknown value (?):

```
"lift line skiing", 1, ? , 617-9666673, "boston"
"off the wall",     2, ? , 781-5555554, "new york"
"flying fat",       3, ? , 303-9992222, "washington"
```

When exporting LONGCHAR values:

- Only one LONGCHAR variable can be exported per command
- Code page information is placed at the beginning of the file

Loading large objects

When you use IMPORT to load large object data into a database or temp-table, you use the INPUT FROM statement to read .blb files and place the file's contents in the BLOB or CLOB fields. You use the optional LOB-DIR*dir-name* phrase on the INPUT FROM statement to specify where the .blb files are located. If you do not specify LOB-DIR, IMPORT loads BLOB and CLOB values from the same directory as the INPUT file. When you import a .blb file into a CLOB, the filename **must** contain the code page name.

During the IMPORT, the AVM interprets the data in the LOB field position in the input file to be the filename for that record's LOB data. In the following example, the input file is called customer.d and the LOB field .blb files are located in C:\DumpedData\custblob:

```
INPUT FROM D:\DumpedData\customer.d LOB-DIR "custblob".
REPEAT:
 CREATE customer:
 IMPORT customer.
END.
INPUT CLOSE.
```

When importing LONGCHAR values:

- Only one LONGCHAR variable can be imported per command
- Code page information must be placed at the beginning of the file
- You can only import files with names that contain code page information

UNDOing subtransactions that contain large object fields

In OpenEdge Releases prior to 10.1A, ABL allowed large object (BLOB and CLOB) operations in subtransactions. However, if the user undid the subtransaction they got the following message:

```
Undo of sub-transaction modifying a large object.  Backing out transaction.
(11599)
```

This message told the user that not just the subtransaction, but the entire transaction was rolled back.

Starting with Release 10.1A, subtransactions containing changes to large object (LOB) fields can be undone without undoing the entire transaction. Users no longer get the previous message and, more importantly, the subtransaction containing the LOB operation is rolled back like any other subtransaction. This feature applies only to LOB operations in the OpenEdge database. The restrictions on LOB operations in subtransactions still applies to the Oracle DataServer.

Note that this feature does **not** apply to temporary tables. Temp-tables must be NO-UNDO if they contain binary (BLOB) or character large object (CLOB) fields.

Passing LONGCHAR variables as parameters

You can use LONGCHAR variables as parameters and return values in procedures, user-defined functions, and methods. ABL passes LONGCHAR parameters by value. This means that the value of the LONGCHAR in memory is copied and passed to the receiving procedure or function. Note that if your application involves passing large LONGCHAR values, this can be an expensive process.

Since LONGCHAR values can be parameters to an Application Server, the CODEPAGE metadata is marshaled and sent along with the data itself. This does not cause a compatibility problem since LONGCHAR is a Release 10 data type and only Release 10 Application Servers and clients recognize it. An error is returned if a LONGCHAR is passed to a Progress Release 9 AppServer. LONGCHAR values are not allowed as return values from a user-defined function or method from the AppServer to the client.

Modifying BLOB data

The following example places a new 4 byte value at a specified offset in the BLOB record. In this example, employee.image is a BLOB field defined for the employee table that contains the employee's picture:

```
DEFINE VAR newValue AS INT.
DEFINE VAR offset AS INT INITIAL 3.
DEFINE VAR m1 AS MEMPTR.

/* Want to update a 4 byte value in the BLOB */
SET-SIZE(m1) = 4.
FIND FIRST employee.
COPY-LOB m1 TO employee.image OVERLAY AT offset.
SET-SIZE(m1) = 0.
```

Using BLOB values and image files

The following example also assumes that there is a new BLOB field (called employee.image) in the employee table that contains the employee's picture. The code reads the BLOB field and writes the data to a temporary file which the LOAD-IMAGE method reads:

```
DEFINE IMAGE PICTURE SIZE 10 BY 5.
DEFINE FRAME frameA picture.

DEFINE VAR picfile AS CHAR.

/* Find the employee record */
FIND FIRST employee WHERE FirstName EQ "John".
/* Create a temporary file to hold the image */
picFile = SESSION:TEMP-DIRECTORY + "\"
   + employee.NAME + ".jpg".
/* Copy the image from the blob field to the
 * temporary file with COPY-LOB code */
COPY-LOB employee.image TO FILE picFile.

/* Associate the image with a button */

picture:LOAD-IMAGE(picFile) IN FRAME frameA.

/* Delete the temporary file */
OS-DELETE VALUE(picFile).
```

Using BLOB values and word processor files

This example assumes that there is a new field (called Report) in the customer table that contains a large MS-Word document. If this field contained an Excel spreadsheet instead of a Word document, the code to retrieve the data would not be different; only the display code would change:

```
DEFINE VAR wordfile AS CHAR INIT "filename".
DEFINE VAR hWordApp AS COM-HANDLE.
FIND FIRST customer.
/*Create a temporary file to hold the Word Document */
wordFile = SESSION:TEMP-DIRECTORY + "\"
    + customer.NAME + ".doc".
/* Copy the image from the blob field to the
 * temporary file with COPY-LOB code */
COPY-LOB customer.Report TO FILE wordFile.
/* Display the Word Document */
CREATE "Word.Application" hWordApp.
hWordApp:VISIBLE = TRUE.
hWordApp:documents:OPEN(wordFile).
```

Creating a CLOB from a file

The following example copies a file to a `CLOB` field. The `COPY-LOB` statement converts the text in `eucjis` to the -cpinternal code page:

```
/*
Assume that fldClob is a CLOB field in a table.
The code page of this file is eucjis.  The CLOB field is in the -cpinternal code page.
*/
DEFINE TEMP-TABLE ttab NO-UNDO
   FIELD name AS CHAR
   FIELD address AS CHAR
   FIELD city AS CHAR
   FIELD state AS CHAR
   FIELD fldClob as CLOB.

CREATE ttab.
COPY-LOB FROM file "clobfile.txt" TO fldClob
CONVERT SOURCE CODEPAGE "eucjis".
```

5

DATETIME and DATETIME-TZ data types

The DATETIME and DATETIME-TZ data types are also new to OpenEdge Release 10. These two ABL data types provide the following benefits (where *datetime* refers to DATETIME or DATETIME-TZ):

- You no longer have to manipulate dates and times separately to have datetime types of data.
- If you use DataServers, you can get direct mapping to their datetime data.
- If you do open ABL programming, you can convert your datetime data directly.

For details, see the following topics:

- DATETIME data type
- DATETIME-TZ data type
- Summary of features
- Related ABL functions
- SESSION handle attributes
- DATETIME and DATETIME-TZ literals
- Comparing date and datetime values
- Display formats
- DATETIME and DATETIME-TZ limits
- Datetime arithmetic
- Passing DATETIME and DATETIME-TZ as parameters

- Dumping and loading DATETIME and DATETIME-TZ fields

DATETIME data type

The DATETIME data type consists of two parts, one an ABL date and one an ABL time. The unit of time is milliseconds from midnight.

The DATETIME data type is useful for dealing with datetime data in the local time of the session that assigned the data, not absolute time. It is also useful for applications that deal with datetime data from one, and only one, time zone.

An example is a procedure that determines peak-time or off-peak times for billing telephone calls. Even though the calls could be made from different time zones, it is the local time of the call that determines the cost of the call.

DATETIME-TZ data type

The DATETIME-TZ data type consists of three parts: an ABL date and time (as for DATETIME), and an integer representing the time zone offset from Coordinated Universal Time (UTC) in minutes.

ABL stores DATETIME-TZ data in UTC, along with the time zone offset of the DATETIME-TZ. This allows for indexing of datetime data based on absolute times. UTC is the current universal standard for time. Local time zone values are relative to UTC (for example, Eastern Standard Time is UTC–05:00). For example, if a DATETIME-TZ field is created in Bedford, MA (time zone offset UTC-05:00) on June 20, 2003 at 15:55, the value stored in the database is 6/20/03 at 20:55 (time converted to milliseconds) with a -5 hour (-300 minute) offset.

Many applications deal with data that spans time zones. The DATETIME-TZ data type is useful for dealing with datetime data in absolute time. Here are some examples of when you might use the DATETIME-TZ data type:

- An order entry application that accepts orders from around the world wants the order datetimes stored in UTC so that they index in absolute time.
- An ABL application communicating with a Web service that includes an XML Schema datetime in its interface can use DATETIME-TZ parameters to communicate in absolute time, and to preserve time zone information from the datetime data coming from the Web service.
- An ABL client/AppServer communication should also use DATETIME-TZ, since the client and the AppServer might be in different time zones. For DATETIME-TZ, the value represents an absolute date and time, and there is no data loss when passing parameters between sessions in different time zones.

Summary of features

The following features apply to the DATETIME and DATETIME-TZ data types:

- ABL supports both DATETIME and DATETIME-TZ in ABL data management statements such as ASSIGN, UPDATE, DISPLAY, IMPORT/EXPORT; by the dynamic BUFFER-FIELD and TEMP-TABLE objects; and by ABL functions and conversions that apply to DATE or TIME.

- You can define DATETIME and DATETIME-TZ database and temp-table fields (static and dynamic). The Data Dictionary supports creating DATETIME and DATETIME-TZ database fields.
- You can define a variable or field AS a DATETIME or DATETIME-TZ variable **or**LIKE a DATETIME or DATETIME-TZ database field.
- You can index DATETIME and DATETIME-TZ database and temp-table fields.
- You can reference DATETIME and DATETIME-TZ data in WHERE clauses.
- You can use all of the ABL constructs (BUFFER-COPY, BUFFER-COMPARE, RAW-TRANSFER) that deal with records containing DATETIME or DATETIME-TZ data.
- You can pass DATETIME and DATETIME-TZ parameters to local or remote procedures and user-defined functions, and as return values from user-defined functions and methods.
- The default initial value for DATETIME and DATETIME-TZ variables or fields is the Unknown value (?). The entire value is unknown if any or all of the parts are unknown.
- You can use NOW to initialize both DATETIME and DATETIME-TZ variable or fields to the current system date and time. The time zone for a DATETIME-TZ initialized to NOW is the time zone of the session creating the DATETIME-TZ.

Related ABL functions

You can use the DATETIME and DATETIME-TZ functions to create DATETIME and DATETIME-TZ variables and fields:

- **DATETIME** — Creates a DATETIME from date and time values or from a character string. The following example uses the DATETIME function:

```
DEF VAR my-datetime as DATETIME.

my-datetime = DATETIME(TODAY, MTIME).

/* The statement above is equivalent to "my-datetime = NOW". */
```

- **DATETIME-TZ** — Creates a DATETIME-TZ from date, time, and time zone values or from a character string. The following example uses the DATETIME-TZ function:

```
DEF VAR my-datetime-tz as DATETIME-TZ.

my-datetime-tz = DATETIME-TZ(TODAY, MTIME, TIMEZONE).

/* The statement above is equivalent to "my-datetime-tz = NOW". */
```

The ABL functions summarized in the following table support the DATETIME and DATETIME-TZ data types. For details on each of these functions, see *OpenEdge Development: ABL Reference*.

Table 6: ABL functions related to DATETIME and DATETIME-TZ

Function	Description
ADD-INTERVAL	Adds or subtracts a specified unit of time (days, months, seconds, etc.) to/from a DATETIME or DATETIME-TZ value. This function takes the date, time, and time zone information into account. You can also use this function with a DATE.
DATE	Returns the date part of DATETIME or DATETIME-TZ.
DAY	Returns the day of the month of the date part of the DATETIME or DATETIME-TZ value.
INTERVAL[1]	Returns an integer that represents an interval between two date/times, expressed in interval-units (days, months, seconds, etc.). This function takes the date, time, and time zone information into account. You can also use this function with a DATE.
ISO-DATE[1]	Returns the character representation of a DATE, DATETIME or DATETIME-TZ that conforms to the ISO 8601 standard for date/time representations - formats are equivalent to the XML Schema date and dateTime formats.
MONTH	Returns an integer value representing the month of the date part of the DATE, DATETIME, or DATETIME-TZ value.
MTIME[1]	Returns an integer representing the time part of a DATETIME or DATETIME-TZ in milliseconds. With no arguments, MTIME returns the number of milliseconds since midnight.
NOW[1]	Returns the current system date, time, and time zone as a DATETIME-TZ. If assigned to a DATETIME, NOW does not include the time zone.
STRING	Converts a value of any data type into a character value.
TIMEZONE[1]	Returns an integer representing the time zone of a DATETIME-TZ in minutes. With no arguments, TIMEZONE returns the current time zone of the session.
WEEKDAY	Returns the weekday of the date part of the DATETIME or DATETIME-TZ value.
YEAR	Returns the year of the date part of the DATETIME or DATETIME-TZ value.

SESSION handle attributes

All of the following SESSION handle attributes apply to the DATETIME and DATETIME-TZ data types.

- **SESSION:DATE-FORMAT (-d)** — Used to set the display format of the date part of DATETIME and DATETIME-TZ; for example, month/day/year (mdy) or year/month/day (ymd).
- **SESSION:YEAR-OFFSET (-yy)** — The start date for the two-digit year range of 100 years; used to display the DATETIMEand DATETIME-TZ when the format specifies a two-digit year.

[1] This ABL function is new to OpenEdge Release 10.

- **SESSION:TIME-SOURCE** — The ABL client/AppServer or database server machine that serves as the time source for the TIME, TODAY, NOW, TIMEZONE, and MTIME functions. This source is also used to generate values for DATE, DATETIME and DATETIME-TZ database and temp-table fields with initial values of TODAY or NOW. The TIME-SOURCE attribute overrides the SESSION:TIMEZONE attribute.
- **SESSION:TIMEZONE** — An INTEGER value that specifies the time zone offset from UTC, in minutes, that ABL uses in the TODAY, TIME, NOW, MTIME and TIMEZONE functions. This source is also used to generate values for DATE, DATETIME and DATETIME-TZ database and temp-table fields with initial values of TODAY or NOW. The SESSION:TIME-SOURCE attribute overrides the TIMEZONE attribute.
- **SESSION:DISPLAY-TIMEZONE** — The time zone for displaying DATETIME-TZ data without a time zone offset.

Precedence of default time zone sources for ABL time date functions

The ABL TODAY, TIME, NOW, MTIME and TIMEZONE functions as well as database and temp-table fields with initial values of TODAY or NOW need a default time zone in order to be correctly resolved by the AVM. The following list describes the different time sources. The first item has the lowest precedence.

- The default time zone of the operating system where the current session is running.
- The time zone offset specified in the TIMEZONE attribute of the SESSION system handle. This attribute can be changed within the session.
- The time zone of the server specified in the TIME-SOURCE attribute of the SESSION system handle. This attribute can be changed within the session.

DATETIME and DATETIME-TZ literals

ABL allows two formats for DATETIME and DATETIME-TZ literals. The formats are the same for both DATETIME and DATETIME-TZ. The first format follows the ISO8601 standard:

```
9999-99-99THH:MM:SS.SSS+HH:MM
```

You can also use the following literal format:

```
"99-99-9999 HH:MM:SS.SSS+HH:MM"
```

The literal must be in quotes because of the space between the date and time. The date part must be in month/day/year order (like DATE literals).

You can omit all or part of the seconds, as well as the time zone offset.

The AVM evaluates the literals at run time. If a time zone offset is specified for a DATETIME literal, the literal is converted to the local time of the session. If a time zone offset is **not** specified for a DATETIME-TZ literal, the DATETIME-TZ uses the session's time zone offset.

Comparing date and datetime values

You cannot compare data of different DATE, DATETIME, and DATETIME-TZ data types to each other using relational operators (for example, EQ) or the MAXIMUM and MINIMUM functions. You must first convert different date and datetime data types to the same data type before doing a comparison between them.

Display formats

The format policy syntax for DATETIME and DATETIME-TZ is similar to the DATE and TIME formats:

Syntax

```
any-date-format   time-with-timezone-format
```

You can leave off part or all of the time format. The date part is subject to the usual -d or SESSION:DATE-FORMAT rules.

The default format for DATETIME is:

```
"99/99/9999 HH:MM:SS.SSS".
```

The default format for DATETIME-TZ is:

```
"99/99/9999 HH:MM:SS.SSS+HH:MM"
```

Valid time formats are:

- "HH"
- "HH:MM"
- "HH:MM:SS"
- "HH:MM:SS.SSS"
- "HH:MM:SS.SSS+HH:MM"
- "HH:MM:SS.SSS AM"
- "HH:MM:SS.SSS AM+HH:MM"

The HH:MM after the plus (+) sign represents time zone offset from UTC. The AVM displays a plus sign (+) when the time zone offset is positive, and a minus sign (-) when it is negative.

The fractional part of seconds is optional (SS, SS.S, SS.SS and SS.SSS are all allowed).

If the time format contains an "A" or "a", instead of displaying the time in 24 hour format, the AVM displays the time in 12 hour format (Hour 0 is 12 am and Hour 12 is 12 pm).

Displaying DATETIME data based on display format

If the format for a DATETIME specifies a time zone, the AVM displays the time zone of the session. For example, consider the following datetime formats:

```
"99/99/99 HH:MM:SS.SSS" "99-99-9999 HH:MM AM" "99-99-9999 HH:MM:SS+HH:MM"
```

March 5, 2002 at 7:15:03.002 am on a client with a time-source time zone of UTC-05:00 would display as (assuming -d mdy):

```
03/05/02 07:15:03.002     03-05-2002 7:15 AM    03-05-2002 07:15:03-05:00
```

Displaying DATETIME-TZ data based on display format

The display of DATETIME-TZ data is affected by the session's time zone offset from UTC. A session's time zone offset is the time zone offset of the SESSION:TIME-SOURCE machine, which is the ABL client/AppServer or database server machine.

If the format for a DATETIME-TZ specifies a time zone offset, the AVM displays the data in the local time of the stored time zone, along with the time zone offset.

If the format does not specify a time zone offset, the AVM displays the DATETIME-TZ in the local time of the session's time zone.

For example, say a DATETIME-TZ field is created in London (time zone UTC+00:00), with a value as above, March 5, 2002 at 7:15:03.002 am.

Using the same formats as above:

```
"99/99/99 HH:MM:SS.SSS"  "99-99-9999 HH:MM AM" "99-99-9999 HH:MM:SS+HH:MM"
```

The ABL client in London (time zone UTC+00:00) displays the field as follows:

```
03/05/02 07:15:03.002     03-05-2002 7:15 AM     03-05-2002 07:15:03+00:00
```

An ABL client in Bedford, MA (time zone UTC-05:00) displays the same field as follows:

```
03/05/02 02:15:03.002     03-05-2002 2:15 AM     03-05-2002 07:15:03+00:00
```

An ABL client in San Diego, CA (time zone UTC-08:00) displays the field as follows:

```
03/04/02 23:15:03.002     03-04-2002 11:15 PM     03-05-2002 07:15:03+00:00
```

If you want to display DATETIME-TZ data without a time zone offset in a time zone other than the session's time zone (a client in Bedford wants to see DATETIME-TZ data in San Diego local time), use the SESSION:DISPLAY-TIMEZONE attribute.

DATETIME and DATETIME-TZ limits

The date part of a DATETIME has the following lower and upper limits: 1/1/32768 B.C. to 12/31/32767 A.D.

The time part has the following limits: 00:00:00.000 to 23:59:61.999.

The seconds can go beyond 60 to account for leap second adjustments (when the atomic clock gets out of sync with the earths revolution). ABL does not account for specific days that contain leap seconds when doing datetime arithmetic. For example, when subtracting two datetimes, the history of any days with leap seconds between those date times is lost—all days between are considered to have exactly 24*60*60*1000 milliseconds).

The time zone part of a DATETIME-TZ must be between -14:00 and +14:00.

Datetime arithmetic

You can use the DATETIME, DATETIME-TZ, INTERVAL, and ADD-INTERVAL functions to add and subtract dates and times in your application.

You can add an integer or decimal to a DATETIME or DATETIME-TZ and get a new DATETIME or DATETIME-TZ. This adds milliseconds to the original DATETIME or DATETIME-TZ. All functions that manipulate DATETIME and DATETIME-TZ data ensure that the time part remains within the valid range, adding a day or days to the date part when the time part goes over the number of milliseconds in a day and subtracting a day or days from the date part when the time part goes under the number of milliseconds in a day.

Adding and subtracting basic datetime values

You can use the addition (+) and subtraction (-) operators with datetime values to obtain results that are defined according to the operation and its operands. So, with the addition operator, you can add milliseconds to a datetime value using the following syntax:

Syntax

```
new-datetime = old-datetime + milliseconds
```

```
new-datetime-tz = old-datetime-tz + milliseconds
```

The result is a different DATETIME or DATETIME-TZ value, respectively.

Syntax

With the subtraction operator, you can subtract milliseconds from a datetime value or subtract a compatible datetime value from a another datetime value using the following syntax:

```
result = datetime - { milliseconds | datetime }
```

```
result = datetime-tz - { milliseconds | datetime-tz }
```

The *result* is either a different DATETIME or DATETIME-TZ value (when subtracting *milliseconds*) or a millisecond interval between two DATETIME or DATATIME-TZ values returned as an INT64.

Adding and subtracting days plus milliseconds with a datetime value

You can add or subtract days and milliseconds together with a datetime value using a combination of ABL functions, depending the data type (DATETIME or DATETIME-TZ). This is the syntax for adding or subtracting a specific number of days and milliseconds with a DATETIME:

Syntax

```
new-datetime = DATETIME( DATE( old-datetime ) { + | - } days ,
   MTIME( old-datetime ) { + | - } milliseconds ) .
```

The *new-datetime* value then represents a new DATETIME at a point later (when adding to) or earlier (when subtracting from) the *old-datetime* value by the specified number of days and milliseconds.

This is the syntax for adding or subtracting a specific number of days and milliseconds with a DATETIME-TZ:

Syntax

```
new-datetime-tz = DATETIME-TZ( DATE( old-datetime-tz ) { + | - } days ,
   MTIME( old-datetime-tz ) { + | - } milliseconds ,
   TIMEZONE( old-dateime-tz ) ) .
```

The *new-datetime-tz* value then represents a new DATETIME-TZ at a point later (when adding to) or earlier (when subtracting from) the *old-datetime-tz* value by the specified number of days and milliseconds within the specified time zone.

Adding, subtracting, and obtaining specific time intervals

The ADD-INTERVAL function adds a time interval to, or subtracts a time interval from, a DATE, DATETIME, or DATETIME-TZ value using the following syntax:

Syntax

```
result = ADD-INTERVAL ( datetime , interval-amount , interval-unit ) .
```

To subtract dates and times, you use the ADD-INTERVAL function with an INTEGER value for the interval amount, and a unit of time (such as minutes, days, or weeks) for the interval unit. The *result* is a different DATE, DATETIME, or DATETIME-TZ value, depending on the data type of *datetime*.

Here is an example that adds two days to the order date to determine the estimated ship date:

```
DEF VAR est-ship-date as DATETIME-TZ.

est-ship-date = ADD-INTERVAL(order-date-tz, 2, "days").
```

Starting with Release 10.1B, you can subtract two datetimes, giving an INT64 with the number of milliseconds.

The INTERVAL function returns the time interval between two DATE, DATETIME, or DATETIME-TZ values using the following syntax:

Syntax

```
result = INTERVAL ( datetime1 , datetime2 , interval-unit ) .
```

The *result* is a the number of specified units (for example, minutes, days, or weeks) between the specified values date or datetime values.

The return data type of INTERVAL is INT64. Here is an example that uses the INTERVAL function to determine if an order shipped on time:

```
/* Find out if an order shipped within 5 days of the order date.
** This example uses the sports2000 database */

FOR EACH Order WHERE Order.CustNum <= 3:
   DISPLAY Order.OrderNum Order.OrderDate Order.ShipDate
      IF INTERVAL(Order.ShipDate, Order.OrderDate, "days") <= 5
         THEN "yes" ELSE "no" LABEL "on-time?".
END.
```

In this example, the ship datetime should be five days after the order date.

The following example using ADD-INTERVAL and INTERVAL shows time-interval calculations with date and datetime values that produce results in milliseconds.

Example using ADD-INTERVAL and INTERVAL

```
&SCOPED-DEFINE vab VIEW-AS ALERT-BOX
OUTPUT TO DATE.out.

DEF VAR d1 AS DATE.
DEF VAR dt1 AS DATETIME.
DEF VAR dt2 AS DATETIME.
DEF VAR dtz1 AS DATETIME-TZ.
DEF VAR dtz2 AS DATETIME-TZ.
DEF VAR i AS int64.
DEF VAR i2 AS INT.
DEF VAR m AS int64.
DEF VAR d AS int64.
DEF VAR y AS int64.
DEF VAR hours AS int64.
DEF VAR minutes AS int64.
DEF VAR seconds AS int64.
DEF VAR millis AS int64.
d1 = 02/13/1945.
dt1 = 1945-02-13T00:59:59.123.
dtz1 = 1945-07-02T23:01:02.123+2:00.

/* ADD-INTERVAL() */
DEF VAR mm AS int64.
mm = 200000000000000.
MESSAGE "pos - ADD-INTERVAL with milliseconds - " ADD-INTERVAL(dtz1, mm,
   "milliseconds") {&vab}.
mm = -9999999999999.
MESSAGE "neg - ADD-INTERVAL with milliseconds - " ADD-INTERVAL(dtz1, mm,
   "milliseconds") {&vab}.
/* INTERVAL() */
dtz1 = 1945-07-30T23:01:02.123+2:00.
dtz2 = 1945-07-01T23:01:02.123+2:00.
MESSAGE "pos DTZ - INTERVAL with milliseconds - " INTERVAL(dtz1, dtz2,
   "milliseconds") {&vab}.
dtz2 = 1945-07-01T23:01:03.123+2:00.
MESSAGE "neg DTZ - INTERVAL with milliseconds - " INTERVAL(dtz1, dtz2,
   "milliseconds") {&vab}.
dt1 = 1945-07-13T00:59:59.123.
dt2 = 1945-02-13T00:59:59.123.
MESSAGE "pos DT - INTERVAL with milliseconds - " INTERVAL(dt1, dt2,
   "milliseconds") {&vab}.
dt2 = 1945-02-13T00:59:59.123.
MESSAGE "neg DT - INTERVAL with milliseconds - " INTERVAL(dt1, dt2,
   "milliseconds") {&vab}.
OUTPUT CLOSE.
```

Passing DATETIME and DATETIME-TZ as parameters

You can pass `DATETIME` and `DATETIME-TZ` fields as parameters between local and remote internal procedures, external procedures, user-defined functions, and methods. This parameter passing is supported across all supported interfaces. However, you should use `DATETIME-TZ` (rather than `DATETIME`) for AppServer and Web services parameters since the client machine and the server machine can be in different time zones. If you use `DATETIME`, you could potentially lose data.

User-defined functions and methods can have `DATETIME` and `DATETIME-TZ` return values.

Dumping and loading DATETIME and DATETIME-TZ fields

You can use EXPORT and IMPORT to dump and load DATETIME and DATETIME-TZ fields. The dump format is fixed and follows the ISO 8601 standard for datetime representations. For DATETIME, there is no time zone offset.

This example shows orders that were created in different time zones:

```
/* dump_order.p */

OUTPUT TO dump.txt.

FOR EACH Order:
   EXPORT SalesRep Carrier OrderDate OrderDateTZ.
END.

OUTPUT CLOSE
```

For example, assume the OrderDate field in the sports2000.Order table is a DATETIME, and there is an additional DATETIME-TZ field, OrderDateTZ that contains the same date and time as OrderDate. Here is the result in dump.txt:

```
"RDR"  "FlyByNight Courier"  2003-11-01T09:15:55.002   2003-11-01T09:15:55.002-05:00
"HXM"  "Walkers Delivery"    2004-01-04T10:24:44.331   2004-01-04T10:24:44.331+00:00
"HXM" "Standard Mail"        2005-03-28T13:02:29.866   2005-03-28T13:02:29.866+02:00
```

6

Arrays

For details, see the following topics:

- Arrays as parameters
- Passing indeterminate arrays as parameters
- Deep copying arrays

Arrays as parameters

Starting with OpenEdge Release 10, you can pass arrays as run-time parameters. Starting in OpenEdge Release 10.2A, you can return arrays from class-based methods and user-defined functions. This feature enables applications developed in ABL to be more compatible with non-ABL applications which expose arrays in their interfaces.

Note: In OpenEdge Release 10.2A, class properties and public data members can now be arrays and defining an instance of a class with the EXTENT option results in an array of handles to that class.

Here is the `DEFINE PARAMETER` statement syntax for defining an array as a parameter:

Syntax

```
DEFINE { INPUT | OUTPUT | INPUT-OUTPUT }
  parameter-name AS data-type EXTENT [expression ] . . .
```

You can define an array parameter as either determinate (has a fixed number of elements) or indeterminate (has an undefined number of elements). To define a determinate array parameter, specify the EXTENT option with the *expression* argument. This optional argument evaluates to an integer value that represents an extent for the array parameter. To define an indeterminate array parameter, specify the EXTENT option without the *expression* argument.

An indeterminate array parameter can be in one of two states: fixed or unfixed, meaning it either has a fixed dimension or it does not. An indeterminate array parameter has an unfixed dimension when first defined. You can fix the dimension of an unfixed indeterminate array parameter by:

- Passing the array parameter to a routine whose corresponding parameter is a determinate array
- Setting the number of elements in the array parameter using the EXTENT statement

ABL treats a fixed indeterminate array parameter as a determinate array parameter; that is, its size is fixed. ABL determines the size of an unfixed indeterminate array parameter at run time.

The EXTENT function returns the size of (the number of elements in) an array field or variable. For example, the following code example returns 12 (the extent of the Mnth-Sales field is 12):

```
FIND FIRST Customer.
DISPLAY EXTENT(Customer.Mnth-Sales).
```

For more information about passing arrays as parameters see *OpenEdge Development: Programming Interfaces*, *OpenEdge Development: ABL Reference*, *OpenEdge Development: Web Services*, *OpenEdge Development: Java Open Clients*, and *OpenEdge Development: .NET Open Clients*.

Passing indeterminate arrays as parameters

An indeterminate array is useful in a reusable routine that can handle any size array.

Consider the following when passing indeterminate array parameters:

- You can pass a determinate array or a fixed indeterminate array to a routine whose corresponding parameter is a determinate array defined as an INPUT, INPUT-OUTPUT, or OUTPUT parameter with the same extent.
- You can pass an unfixed indeterminate array to a routine whose corresponding parameter is another unfixed indeterminate array defined as an INPUT, INPUT-OUTPUT, or OUTPUT parameter.
- You can pass a determinate array or an indeterminate array (fixed or unfixed) to a routine whose corresponding parameter is an unfixed indeterminate array. When passing a determinate or a fixed indeterminate array to an unfixed indeterminate array defined as either an INPUT or INPUT-OUTPUT parameter, the unfixed indeterminate array parameter inherits the fixed dimension from the calling routine (but its definition remains unfixed). When passing a fixed array to an unfixed indeterminate array defined as an OUTPUT parameter, the extents must match upon returning to the calling routine. In this case, the called routine must fix the extent by either applying the EXTENT statement or inheriting the fixed dimension from another called routine.

- You can pass an unfixed indeterminate array to a routine whose corresponding parameter is a determinate array defined as an OUTPUT or INPUT-OUTPUT parameter. In this case, the unfixed indeterminate array inherits the fixed dimension from the called routine. Likewise, an unfixed indeterminate array inherits the dimension of an indeterminate array parameter fixed using the EXTENT statement in the called routine. You cannot pass an unfixed indeterminate array to a routine whose corresponding parameter is a determinate array defined as an INPUT parameter.
- You cannot pass an unfixed indeterminate array to a COM object, DLL routine, or UNIX shared library routine.

The following example passes a determinate array as an input parameter to a routine that defines the parameter as an unfixed indeterminate array:

```
DEFINE VARIABLE x AS INTEGER EXTENT 3.

RUN foo (INPUT x).

PROCEDURE foo:
    DEFINE INPUT PARAMETER x AS INTEGER EXTENT.
    MESSAGE EXTENT(x). /* Returns 3 */
END.
```

In this case, the unfixed indeterminate array parameter inherits the fixed dimension from the calling routine (but its definition remains unfixed).

The following example passes an unfixed indeterminate array as an output parameter to a routine that defines the parameter as a determinate array:

```
DEFINE VARIABLE x AS INTEGER EXTENT.

MESSAGE EXTENT(x). /* Returns ? */
RUN foo (OUTPUT x).
MESSAGE EXTENT(x). /* Returns 4 */

PROCEDURE foo:
    DEFINE OUTPUT PARAMETER x AS INTEGER EXTENT 4.
END.
```

In this case, the unfixed indeterminate array parameter inherits a fixed dimension from the called routine.

The following example passes an unfixed indeterminate array as an output parameter to a routine that also defines the parameter as an unfixed indeterminate array, but fixes the dimension of the array using the EXTENT statement:

```
DEFINE VARIABLE x AS INTEGER EXTENT.

MESSAGE EXTENT(x). /* Returns ? */
RUN foo (OUTPUT x).
MESSAGE EXTENT(x). /* Returns 4 */

PROCEDURE foo:
   DEFINE OUTPUT PARAMETER x AS INTEGER EXTENT.
   EXTENT(x) = 4. /* Fixes the extent to 4 */
END.
```

In this case, the unfixed indeterminate array parameter also inherits a fixed dimension from the called routine.

Deep copying arrays

Traditionally, you copy one array to another array using a loop or nested loops to copy one array element at a time. In Release 10.2A, you can also use an ASSIGN statement to deep copy (clone) one array to another, where it makes semantic sense to do so. This process copies all elements of one array into memory and then copies the data to the target array. Therefore, deep copying may not be practical for very large arrays. For deep copies, use an ASSIGN statement and unsubscripted references to the array object. For example:

```
DEFINE VARIABLE firstArray AS INTEGER EXTENT 3.
DEFINE VARIABLE secondArray AS INTEGER EXTENT 3.

ASSIGN firstArray[1] = 100
       firstArray[2] = 200
       firstArray[3] = 300.

ASSIGN secondArray = firstArray.
```

When deep copying one array to another, the following rules apply:

- If both the array on the left-hand side and the right-hand side of the assignment are determinate arrays, the EXTENT size must match or the AVM raises an error.
- You cannot assign an indeterminate array to a determinate array. That is, if an array does not yet have a fixed size, you cannot assign it to one that does have a fixed size.
- You can assign any array to an indeterminate array and the indeterminate array is fixed at the size of the other array.
- You cannot assign a scalar value to an indeterminate array.

For more information, see the ASSIGN statement reference entry in *OpenEdge Development: ABL Reference*.