Progress®



Use Dynamic Call Objects in ABL Applications

OpenEdge®

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

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

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

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

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

ABL elements related to the call object.....9

Attributes of a call object handle.....9
Methods of a call object handle.....12

When to use the call object.....15

Code examples.....17

Run an internal procedure of a persistent external procedure.....18
Run an internal procedure of a single-run or singleton procedure.....19
Use a call object multiple times.....19
Get an attribute.....20
Set an attribute.....21
Multiple dynamic invokes from a temp-table.....21
Invoke a Windows DLL routine.....22
Implement a sleep timer.....23

Preface

Purpose

This set of topics describes how to:

- Dynamically execute a procedure
- Execute a user-defined function
- Get or set a handle attribute
- Invoke a handle method
- Invoke a Windows Dynamic Link Library (DLL) routine or invoke a UNIX shared library routine using the call object handle

---

Note: In the context of this set of topics, *method* refers to a built-in, handle-based ABL method, such as `qryhandle:GET-NEXT( )`. You cannot run a class-based method using this feature. You can, however, dynamically invoke a class-based method using the `DYNAMIC-INVOKE` function or the `Invoke( )` method of the `Progress.Lang.Class` class. For more information, see the *ABL Reference.*

Audience

These topics are intended for all ABL developers.

Organization

The following sections are included:

- ABL elements related to the call object on page 9
- When to use the call object on page 15
- Code examples on page 17

Documentation conventions

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

Purpose

This set of topics describes how to:

- Dynamically execute a procedure
- Execute a user-defined function
- Get or set a handle attribute
- Invoke a handle method
- Invoke a Windows Dynamic Link Library (DLL) routine or invoke a UNIX shared library routine using the call object handle

---

Note: In the context of this set of topics, *method* refers to a built-in, handle-based ABL method, such as `qryhandle:GET-NEXT( )`. You cannot run a class-based method using this feature. You can, however, dynamically invoke a class-based method using the `DYNAMIC-INVOKE` function or the `Invoke( )` method of the `Progress.Lang.Class` class. For more information, see the *ABL Reference.*

Audience

These topics are intended for all ABL developers.

Organization

The following sections are included:

- ABL elements related to the call object on page 9
- When to use the call object on page 15
- Code examples on page 17

Documentation conventions

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

1

ABL elements related to the call object

The ABL elements related to the call object consist of a statement, a handle, attributes, and methods.

Note: For more information on these ABL elements, see *ABL Reference*.

The CREATE CALL statement creates a call object, then stores a handle to it in the handle variable you specify. A call object handle lets you do the following dynamically:

- Invoke an external procedure, internal procedure, or user-defined function
- Invoke a Windows DLL routine or UNIX shared library routine
- Get or set a handle attribute
- Run a handle method

For details, see the following topics:

- Attributes of a call object handle
- Methods of a call object handle

Attributes of a call object handle

The following table lists the attributes of the call object handle.

Table 1: Attributes of a call object handle

Attribute	Type	Description
ASYNCHRONOUS attribute	LOGICAL	Applies only when the logic to be dynamically invoked resides on a Progress AppServer™.Indicates if you want the logic to run asynchronously.
ASYNC-REQUEST-HANDLE attribute	HANDLE	Applies only when ASYNCHRONOUS is TRUE. A handle to an asynchronous-request object containing detailed information on your asynchronous request. Set by the INVOKE( ) method when ASYNCHRONOUS is TRUE.
CALL-NAME attribute	CHARACTER	The name of one of the following: • A procedure or user-defined function you want to invoke dynamically • An attribute you want to get or set dynamically • A method you want to invoke dynamically
CALL-TYPE attribute	INTEGER	Indicates if you are dynamically: • Invoking a procedure • Invoking a user-defined function • Invoking a Windows DLL or UNIX shared library routine • Getting an attribute or invoking a method • Setting an attribute
EVENT-PROCEDURE attribute	CHARACTER	Applies only when ASYNCHRONOUS is TRUE. The name of an internal procedure you want executed when a dynamic, asynchronous invoke completes. Note: You must set EVENT-PROCEDURE before you execute INVOKE( ).

Attribute	Type	Description
EVENT-PROCEDURE-CONTEXT attribute	HANDLE	Applies only when ASYNCHRONOUS is TRUE. A handle to a running persistent procedure containing an internal procedure you want executed when a dynamic, asynchronous invoke completes.
HANDLE attribute	HANDLE	A handle to the object.
IN-HANDLE attribute	HANDLE	Contains one of the following: • A handle to a persistent, single-run, or singleton procedure • A handle to an object, such as a buffer or socket, whose attributes or methods you want to invoke dynamically
INSTANTIATING-PROCEDURE attribute	HANDLE	Returns the handle to the procedure in which an object was instantiated.
IS-PARAMETER-SET attribute	LOGICAL	Indicates if you have already set a particular parameter.
LIBRARY attribute	CHARACTER	Specifies the name of a Windows DLL or a UNIX shared library.
LIBRARY-CALLING-CONVENTION attribute	CHARACTER	Specifies the calling convention for a Windows DLL or a UNIX shared library.
NUM-PARAMETERS attribute	INTEGER	The number of parameters you are passing.
ORDINAL attribute	INTEGER	Specifies the number of the entry point of the Windows DLL routine to invoke.
PERSISTENT attribute	LOGICAL	Indicates if you want an external procedure you plan to invoke dynamically to run as persistent.[1]
PROCEDURE-TYPE attribute	CHARACTER	Indicates if you want an external procedure you plan to invoke dynamically to run as persistent, single-run, or singleton. Takes values of "PERSISTENT", "SINGLE-RUN", or "SINGLETON", respectively.[1]

[1] See Run an internal procedure of a persistent external procedure on page 18 for details on how the PERSISTENT and PROCEDURE-TYPE attributes interact.

Attribute	Type	Description
RETURN-VALUE attribute	RETURN-VALUE-DATA-TYPE	Applies only to user-defined functions, attributes, and methods.Contains one of the following: • The value returned by a user-defined function • The value of an attribute • The value returned by a method
RETURN-VALUE-DATA-TYPE attribute	CHARACTER	The data type of the return value. Applies only to user-defined functions, attributes, and methods. If you set this before you execute the INVOKE( ) method, the return value is converted to this data type. If you do not set this before you execute the INVOKE( ) method, it indicates the return value's data type.
RETURN-VALUE-DLL-TYPE attribute	CHARACTER	Specifies the expected data type returned from a Windows DLL routine.
SERVER attribute	HANDLE	A handle to an AppServer containing logic you want to invoke dynamically.
TYPE attribute	CHARACTER	The type of a handle.

Methods of a call object handle

The following table lists the methods of a call object handle.

Table 2: Methods of a call object handle

Method	Return Type	Description
CLEAR( ) method	LOGICAL	Resets each attribute of the call object to its default value. **Note:** To reset just the parameters, set the NUM-PARAMETERS attribute to zero.

Method	Return Type	Description
INVOKE( ) method	None (similar to the RUN statement)	Lets you dynamically do the following: • Invoke an external procedure, internal procedure, or user-defined function • Get or set an attribute • Invoke a method **Note:** INVOKE( ) cannot occur within an expression.
SET-PARAMETER( ) method	LOGICAL	Lets you set parameters for one of the following: • A procedure or user-defined function you want to invoke dynamically • An attribute you want to get or set dynamically • A method you want to invoke dynamically • A Windows DLL routine or a UNIX shared library routine you want to invoke dynamically.

2

When to use the call object

For some use cases, programming with the call object requires more lines of code than other ABL alternatives. The following table shows an example of the code required to invoke an external procedure dynamically and statically.

Table 3: Invoking hello.p dynamically and statically

Dynamic invoke	Static invoke

`DEFINE VARIABLE hCall AS HANDLE`
`  NO-UNDO.`
 	
`CREATE CALL hCall.`
 	
`/* Invoke hello.p non-persistently */`
`ASSIGN`
`  hCall:CALL-NAME      = "hello.p"`
`  hCall:NUM-PARAMETERS = 1.`
 	
`hCall:SET-PARAMETER (1, "CHARACTER",`
`  "INPUT", "Hello world").`
 	
`hCall:INVOKE( ).`
 	
`/* Clean up */`
`DELETE OBJECT hCall.`
	`RUN hello.p ("Hello world").`

As the previous table illustrates, executing `hello.p` dynamically using the call object requires many more lines of code and is therefore less efficient than doing it with static invoke.

Consider using the call object for these situations:

- To invoke an internal or external procedure whose *calling sequence* (number of parameters and the data type of each) is unknown at compile time.

  Note: If the only the **name** of the procedure is unknown at compile time, use the `RUN` statement with the `VALUE` option, and avoid using the call object.

- To invoke a function whose calling sequence is unknown at compile time

  Note: If the only the **name** of the function is unknown at compile time, use the `DYNAMIC-FUNCTION( )` function, and avoid using the call object.

- To reference a widget attribute or method whose name is unknown at compile time.

  If you already know the name of the attribute or procedure, then you also know its syntax, since the name implies certain syntax. If you know the syntax, then you know the calling sequence, since the syntax defines the calling sequence. If you know the calling sequence, you can use widget:attribute or widget:method syntax, and avoid using the call object.

- To dynamically invoke a Windows DLL routine or a UNIX shared library routine when the following is true:
  - The number of parameters and their data type is only known at run time
  - The routine exists in both a Windows DLL and a UNIX shared library
  - The routine has a variable number of parameters

3

Code examples

The next set of topics contain code examples showing some ways in which the call object can be used.

- Run an internal procedure of a persistent external procedure
- Run an internal procedure of a single-run or singleton procedure
- Use a call object multiple times
- Get an attribute
- Set an attribute
- Multiple dynamic invokes from a temp-table
- Invoke a Windows DLL routine
- Implement a sleep timer

For details, see the following topics:

- Run an internal procedure of a persistent external procedure
- Run an internal procedure of a single-run or singleton procedure
- Use a call object multiple times
- Get an attribute
- Set an attribute
- Multiple dynamic invokes from a temp-table
- Invoke a Windows DLL routine

- Implement a sleep timer

Run an internal procedure of a persistent external procedure

The following example, which illustrates using a single call object multiple times, is a bit more complicated than that shown in Attributes of a call object handle on page 9. This example also shows how to:

- Dynamically invoke the external procedure `persis.p` persistently
- Dynamically invoke an internal procedure of `persis.p`, `internal-persis-proc`, with an input parameter of type "`INTEGER`" and with the value 333

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

CREATE CALL hCall.

/* Invoke persis.p persistently */
ASSIGN
  hCall:CALL-NAME  = "persis.p"
  /* Sets CALL-TYPE to the default */
  hCall:CALL-TYPE  = PROCEDURE-CALL-TYPE
  hCall:PERSISTENT = TRUE.

hCall:INVOKE.

/* Invoke internal-persis-proc in persis.p */
ASSIGN
  hCall:CALL-NAME      = "internal-persis-proc"
  /* Sets CALL-TYPE to the default */
  hCall:CALL-TYPE      = PROCEDURE-CALL-TYPE
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "INTEGER", "INPUT", 333).
hCall:INVOKE( ).

/* Clean up */
DELETE PROCEDURE hCall:IN-HANDLE.
DELETE OBJECT hCall.
```

When persis.p is invoked dynamically, the handle of the running persistent procedure is stored automatically in the call object's `IN-HANDLE` attribute. When `internal-persis-proc` is invoked dynamically, ABL knows it resides in the running persistent procedure whose handle is stored in the call object's `IN-HANDLE` attribute.

You can also run a procedure persistently by setting the `PROCEDURE-TYPE` attribute to "`PERSISTENT`". Setting the `PROCEDURE-TYPE` attribute to "`PERSISTENT`" and setting the `PERSISTENT` attribute to "`TRUE`" are equivalent, and setting one will automatically set the other. Setting the two attributes to conflicting values, e.g., `PROCEDURE-TYPE` to "`SINGLETON`" and `PERSISTENT` to `TRUE`, will cause a run-time error.

Run an internal procedure of a single-run or singleton procedure

The following example shows how to dynamically invoke a procedure as single-run and then dynamically invoke an internal procedure of the single-run. The same syntax can be used to call a procedure as persistent or singleton by setting PROCEDURE-TYPE to "PERSISTENT" or "SINGLETON", respectively.

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

CREATE CALL hCall.

/* Invoke single.p as single-run*/
ASSIGN
  hCall:CALL-NAME  = "single.p"
  /* Sets CALL-TYPE to the default */
  hCall:CALL-TYPE  = PROCEDURE-CALL-TYPE
  hCall:PROCEDURE-TYPE = SINGLE-RUN.

hCall:INVOKE.

/* Invoke internal-single-proc in single.p */

ASSIGN
  hCall:CALL-NAME      = "internal-single-proc"
  /* Sets CALL-TYPE to the default */
  hCall:CALL-TYPE      = PROCEDURE-CALL-TYPE
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "INTEGER", "INPUT", 333).
hCall:INVOKE( ).

/* Clean up */
DELETE PROCEDURE hCall:IN-HANDLE.
DELETE OBJECT hCall.
```

Use a call object multiple times

This example further demonstrates using a call object multiple times by:

1. Dynamically invoking the procedure file hello.p with the "CHARACTER" input parameter, string "HELLO WORLD"
2. Dynamically invoking the external procedure persis.p persistently
3. Dynamically invoking an internal procedure of persis.p, internal-persis-proc, with an input parameter of type "INTEGER" and with the value 333

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

 CREATE CALL hCall.

/* Invoke hello.p nonpersistently */
ASSIGN
  hCall:CALL-NAME      = "hello.p"
```

```
  /* Set CALL-TYPE to the default */
  hCall:CALL-TYPE      = PROCEDURE-CALL-TYPE
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "CHARACTER", "INPUT", "HELLO WORLD").
hCall:INVOKE( ).

/* Reset the call object handle */
hCall:CLEAR( ).

/* Invoke persis.p persistently */
ASSIGN
  hCall:CALL-NAME  = "persis.p"
  /* Set CALL-TYPE to the default */
  hCall:CALL-TYPE  = PROCEDURE-CALL-TYPE
  hCall:PERSISTENT = TRUE.

hCall:INVOKE.

/* Invoke internal-persis-proc in persis.p */
ASSIGN
  hCall:CALL-NAME      = "internal-persis-proc"
  /* Set CALL-TYPE to the default */
  hCall:CALL-TYPE      = PROCEDURE-CALL-TYPE
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "INTEGER", "INPUT", 333).
hCall:INVOKE( ).

/* Clean up */
DELETE PROCEDURE hCall:IN-HANDLE.
DELETE OBJECT hCall.
```

This example resets the call object handle by using the `CLEAR( )` method between invoking `hello.p` (the first invoke) and invoking `persis.p` (the second invoke).

Get an attribute

This example gets the current value of the `TITLE` attribute of a frame. The call object's `CALL-TYPE` attribute is set to `GET-ATTR-CALL-TYPE`. The value of the `TITLE` attribute is returned through the call object's `RETURN-VALUE` attribute, as shown:

```
/* Get title of frame */

ASSIGN
  hCall:IN-HANDLE = myframe_handle
  hCaLL:CALL-TYPE = GET-ATTR-CALL-TYPE
  hCall:CALL-NAME = "TITLE".

hCall:INVOKE( ).
Mytitle = hCall:RETURN-VALUE.
```

Set an attribute

This example sets the SESSION handle's NUMERIC-FORMAT attribute to "european":

```
/* Set SESSION:NUMERIC-FORMAT to "european" */

ASSIGN
  hCall:IN-HANDLE      = "session"
  hCall:CALL-TYPE      = SET-ATTR-CALL-TYPE
  hCall:CALL-NAME      = "numeric-format"
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "CHARACTER", "INPUT", "european").
hCall:INVOKE( ).
```

In contrast to the Get an attribute on page 20 example, this "Setting an attribute" example:

- Sets the CALL-TYPE attribute to SET-ATTR-CALL-TYPE
- Sets the NUM-PARAMETERS attribute to 1
- Runs the SET-PARAMETER( ) method to make the first parameter an INPUT parameter of data type CHARACTER with the value "european"

Multiple dynamic invokes from a temp-table

This example shows how a series of dynamic invokes of the call object can be set up using a temp-table, ttParam, whose handle is passed into the procedure:

```
/* ttParam is a temp-table that has one record with these fields:
   parm_1
   parm_2
   ...
   parm_n
   run-name
   nparms
   datatypes, extent nparms
   iomodes, extent nparms
*/

DEFINE INPUT PARAMETER TABLE-HANDLE ttParam NO-UNDO.

DEFINE VARIABLE hCall    AS HANDLE  NO-UNDO.
DEFINE VARIABLE hDtypes  AS HANDLE  NO-UNDO.
DEFINE VARIABLE hIOmodes AS HANDLE  NO-UNDO.
DEFINE VARIABLE ix       AS INTEGER NO-UNDO.

ASSIGN
  hDtypes  = ttParam:BUFFER-FIELD("datatypes")
  hIOmodes = ttParam:BUFFER-FIELD("iOmodes").

ttParam:FIND-FIRST.

CREATE CALL hCall.
ASSIGN
```

```
  hCall:CALL-NAME      = ttParam:BUFFER-FIELD("run-name"):BUFFER-VALUE
  hCall:NUM-PARAMETERS = ttParam:BUFFER-FIELD("nparms"):BUFFER-VALUE.

FOR ix = 1 TO hCall:NUM-PARAMETERS:
  hCall:SET-PARAMETER(ix, hDtypes:BUFFER-VALUE(ix),
    hIOmodes:BUFFER-VALUE(ix), ttParam:BUFFER-FIELD(ix):BUFFER-VALUE).
END.

hCall:INVOKE( ).
DELETE OBJECT hCall.
```

To implement this example, you must:

1. Define a temp-table record structure where each field represents one data item involved in dynamic invoke
2. Load the temp-table with a row of data for each dynamic invoke to be performed
3. Read the temp-table and performing a dynamic invoke for every record

Setting up a temp-table in this way allows an application to perform any number of dynamic invokes on the fly.

Invoke a Windows DLL routine

The following example demonstrates the use of the RETURN-VALUE-DLL-TYPE attribute and the RETURN-VALUE attribute when invoking a Windows DLL routine. RETURN-VALUE-DLL-TYPE is set to "LONG", which is the value that the DLL routine expects to receive. For example:

```
FUNCTION GetWinVersion RETURNS INTEGER:
  DEFINE VARIABLE cValue  AS CHARACTER NO-UNDO.
  DEFINE VARIABLE libName AS CHARACTER NO-UNDO.
  DEFINE VARIABLE hCall   AS HANDLE    NO-UNDO.

  CREATE CALL hCall.
  ASSIGN
    hCall:CALL-NAME             = "GetVersion"
    hCall:LIBRARY               = "kernel32.dll"
    hCall:CALL-TYPE             = DLL-CALL-TYPE
    hCall:RETURN-VALUE-DLL-TYPE = "LONG".

  hCall:INVOKE( ).
  cValue = hCall:RETURN-VALUE.

  DELETE OBJECT hCall.
  RETURN cValue.
END FUNCTION.
```

After the invoke, RETURN-VALUE contains the "INTEGER" data type.

Implement a sleep timer

The following example implements the user-defined function `sleep`, which causes the AVM to sleep for a specified number of milliseconds:

```
FUNCTION sleep RETURNS INTEGER (msecs AS INTEGER):
  DEFINE VARIABLE cFunction AS CHARACTER NO-UNDO INITIAL "sleep".
  DEFINE VARIABLE cLibrary  AS CHARACTER NO-UNDO INITIAL "libc.so.1".
  DEFINE VARIABLE hCall     AS HANDLE    NO-UNDO.

  CREATE CALL hCall.
  ASSIGN
    cLibrary             = "kernel32.dll" WHEN OPSYS = "WIN32"
    cFunction            = "Sleep" WHEN OPSYS = "WIN32"
    hCall:CALL-NAME      = cFunction
    hCall:LIBRARY        = cLibrary
    hCall:CALL-TYPE      = DLL-CALL-TYPE
    hCall:NUM-PARAMETERS = 1.

  hCall:SET-PARAMETER(1, "LONG", "INPUT", msecs).
  hCall:INVOKE( ).

  DELETE OBJECT hCall.
  RETURN msecs.
END FUNCTION.
```

Note that the code determines on which OS it is running, and invokes the appropriate Windows DLL or UNIX shared library.