Progress®



Use the Progress OpenEdge JMS Adapter

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: 2021/06/29

Table of Contents

Preface....15

Learn about the Progress OpenEdge JMS Adapter....17

Prerequisites for using the Progress OpenEdge JMS Adapter....18
Main steps for using the Progress OpenEdge JMS Adapter....18

Install the Progress OpenEdge JMS Adapter....19

Install the OpenEdge JMS Adapter interactively....19
Install the OpenEdge JMS adapter on UNIX....29
Install the OpenEdge JMS Adapter in silent mode....35

Configure the Progress OpenEdge JMS Adapter....39

Configure your JMS provider....39
Use the Progress OpenEdge JMS Adapter and the ABL-JMS API with JMS Administered Objects....42
Build the AdminObjectFinder for use with JNDI....42
AdminObjectFinder.java example for IBM WebsphereMQ....43
AdminObjectFinder.java and jndi.properties example for ActiveMQ Artemis....45
Start the Progress OpenEdge JMS Adapter....47
Configure logging for the Progress OpenEdge JMS adapter....48

Use the ABL-JMS Producer and Consumer APIs....49

Select the JMS Adapter connection mode....50
Progress OpenEdge JMS Adapter BrokerConnect....50
Progress OpenEdge JMS Adapter ClientConnect....52
Manage connections and sessions....54
Create a JMS session....54
Delete a JMS session....55
Connection options....55
Set and get JMS connection and session attributes....57
Connect to the Progress OpenEdge JMS Adapter 58
Establish session control....58
Access message delivery parameters....58
Request/Reply....59
Work with messages....60
Manage the lifecycle of an ABL - JMS message....60
Create, populate, and access messages....61
Send messages to a queue....69
Publish messages to a topic....69

Clear messages....70
Delete messages....70
Access message header properties....70
Access message properties....71
Consume messages....71
Create a Message Consumer object....72
Create a message handler process....72
Set reply properties....73
Receive messages from a queue....73
Subscribe to a topic....73
Terminate the Message Consumer object....73
Process messages....74
Control flow of messages....74
Reuse messages....74
Message-reception issues....74
Reply mechanisms....75
Transacted sessions....76
Message acknowledgement, forwarding, and recovery....77
Error and condition handling....79
Handling errors....79
Synchronously reported errors and conditions....80
Asynchronously reported conditions....80
Run-time conditions....81
Connection and communication failures....81
Message handler errors and conditions....82
Interrupts....82
Progress OpenEdge JMS Adapter failure....82
PTP message example....82
PTP message example summary....84
Pub/Sub messaging example....85
Pub/sub messaging example summary....87
Programming scenarios....88
Using JMS 1.1 unified domain model....88
Implement ClientConnect....89
Enhanced XML support....89
TempTableMessage....96
DataSetMessage....99

Configure JMS-provider proprietary features....103

Configure fault tolerance and high availability for ActiveMQ Artemis....104
Access JMS Administered Objects for SonicMQ....110
Create a serialized connection object....111
Find JMS Administered Objects in JNDI or proprietary directories....115
Connection file parameters....117

Configure client persistence for SonicMQ....119
Client persistence example for SonicMQ....120
Configure fault tolerance for SonicMQ....121
Fault tolerance examples for SonicMQ....122
Load balance SonicMQ brokers....123
Manage fail-over support for SonicMQ....123
Configure message selectors for SonicMQ....124

ABL-JMS API Reference....125

Session objects....132
jmssession.p....132
ptpsession.p....133
pubsubsession.p....133
Methods in the Session objects....134
Methods in the Message Consumer objects....136
Methods in the Message objects....137
SonicMQ proprietary APIs....139
acknowledgeAndForward procedure....141
addMessagePart procedure....142
createChangeStateConsumer procedure....143
createMultipartMessage procedure....144
createRejectedMessageConsumer procedure....144
getBytesPartByID function....145
getBytesPartByIndex function....145
getClientPersistence function....146
getClientTransactionBufferSize function....147
getConnectionURLs function....147
getContentType....148
getFaultTolerant function....148
getFaultTolerantReconnectTimeout function....149
getFlowToDisk function....149
getInitialConnectionTimeout function....150
getLocalStoreDirectory function....151
getLocalStoreSize function....151
getLocalStoreWaitTime function....152
getReconnectInterval function....152
getReconnectTimeout function....153
getSelectorAtBroker function....154
getSequential function....154
getSingleMessageAcknowledgement function....155
isFaultTolerant function....155
setClientPersistence procedure....156
setClientTransactionBufferSize procedure....157
setConnectID procedure....157

setConnectionFile procedure....158
setConnectionURLs procedure....158
setFaultTolerant procedure....159
setFaultTolerantReconnectTimeout procedure....160
setFlowToDisk procedure....161
setInitialConnectionTimeout procedure....161
setLocalStoreDirectory procedure....162
setLocalStoreSize procedure....163
setLocalStoreWaitTime procedure....164
setPingInterval procedure....164
setPrefetchCount procedure....165
setPrefetchThreshold procedure....166
setReconnectInterval procedure....167
setReconnectTimeout procedure....167
setSelectorAtBroker procedure....168
setSequential procedure....168
setSingleMessageAcknowledgement procedure....169
addBytesPart procedure....170
addTextPart procedure....170
appendText procedure....171
beginSession procedure....172
browseQueue procedure....172
cancelDurableSubscription procedure....173
clearBody procedure....174
clearProperties procedure....174
commitReceive procedure....175
commitSend procedure....175
createBytesMessage procedure....176
createDataSetMessage procedure....176
createHeaderMessage procedure....177
createMapMessage procedure....177
createMessageConsumer procedure....178
createStreamMessage procedure....179
createTemporaryQueue procedure....179
createTemporaryTopic procedure....180
createTempTableMessage procedure....180
createTextMessage procedure....181
createXMLMessage procedure....181
deleteConsumer procedure....182
deleteMessage procedure....182
deleteSaxWriter procedure....183
deleteSession procedure....183
deleteTemporaryQueue procedure....184
deleteTemporaryTopic procedure....184
endOfStream function....185

getAdapterService function....186
getApplicationContext function....186
getBrokerURL function....187
getBytesCount function....187
getBytesToRaw function....188
getChar function....188
getCharCount function....189
getCharProperty function....189
getClientID function....190
getConnectID function....190
getConnectionID function....191
getConnectionMetaData function....191
getDataSet function....192
getDate function....192
getDateProperty function....193
getDateTime function....193
getDateTimeProperty function....194
getDateTime-TZ function....195
getDateTimeTzProperty function....195
getDecimal function....196
getDecimalProperty function....196
getDefaultPersistency function....197
getDefaultPriority function....197
getDefaultTimeToLive function....198
getDestinationName function....198
getInt function....199
getIntProperty function....199
getInt64 function....200
getInt64Property function....200
getItemType function....201
getJMSCorrelationID function....201
getJMSCorrelationIDAsBytes function....202
getJMSDeliveryMode function....202
getJMSDestination function....203
getJMSExpiration function....203
getJMSMessageID function....204
getJMSPriority function....204
getJMSRedelivered function....205
getJMSReplyTo function....205
getJMSServerName function....206
getJMSTimestamp function....206
getJMSType function....207
getLoadBalancing function....207
getLogical function....208
getLogicalProperty function....208

getLongString function....209
getLongStringCP function....209
getLongText function....210
getLongTextCP function....210
getMapNames function....211
getMemptr function....212
getMessagePartByID function....212
getMessagePartByIndex function....213
getMessageType function....214
getNoAcknowledge function....214
getPartCount function....215
getPassword....215
getProcHandle function....216
getProcName function....216
getPropertyNames function....216
getPropertyType function....217
getReplyAutoDelete function....217
getReplyPersistency function....218
getReplyPriority function....218
getReplyTimeToLive function....219
getReplyToDestinationType function....219
getReuseMessage function....220
getSaxWriter function....221
getSession function....221
getShutdownWaitFor function....222
getTempTable function....222
getText function....223
getTextPartByID function....223
getTextPartByIndex function....224
getTextSegment function....224
getTransactedReceive function....225
getTransactedSend function....225
getUser function....226
getX-Document function....226
hasReplyTo function....227
inErrorHandling function....227
inMessageHandling function....228
inQueueBrowsing function....228
inReplyHandling function....229
isMessagePart function....229
JMS-MAXIMUM-MESSAGES global variable....230
messageHandler procedure....230
moveToNext procedure....231
publish procedure....232
readBytesToRaw procedure....233

readChar function....234
readDate function....234
readDateTime function....235
readDateTime-TZ function....236
readDecimal function....236
readInt function....237
readInt64 function....238
readLogical function....238
readLongString function....239
readLongStringCP function....239
receiveFromQueue procedure....240
recover procedure....241
requestReply procedure....241
reset procedure....242
rollbackReceive procedure....243
rollbackSend procedure....243
sendToQueue procedure....244
setAdapterService procedure....245
setApplicationContext procedure....245
setBoolean procedure....246
setBooleanProperty procedure....247
setBrokerURL procedure....247
setByte procedure....248
setByteProperty procedure....248
setBytesFromRaw procedure....249
setChar procedure....250
setClientID procedure....250
setDataSet procedure....251
setDate procedure....251
setDateProperty procedure....252
setDateTime procedure....253
setDateTimeProperty procedure....254
setDateTime-TZ procedure....254
setDateTimeTzProperty procedure....255
setDefaultPersistency procedure....256
setDefaultPriority procedure....256
setDefaultTimeToLive procedure....257
setDouble procedure....257
setDoubleProperty procedure....258
setErrorHandler procedure....258
setFloat procedure....259
setFloatProperty procedure....260
setInt procedure....260
setIntProperty procedure....261
setInt64 procedure....261

setInt64Property procedure....262
setJMSCorrelationID procedure....262
setJMSCorrelationIDAsBytes procedure....263
setJMSReplyTo procedure....263
setJMSServerName procedure....264
setJMSType procedure....265
setLoadBalancing procedure....265
setLong procedure....266
setLongProperty procedure....266
setLongString procedure....267
setLongText procedure....267
setMemptr procedure....268
setNoAcknowledge procedure....269
setNoErrorDisplay procedure....269
setPassword procedure....270
setReplyAutoDelete procedure....271
setReplyPersistency procedure....271
setReplyPriority procedure....272
setReplyTimeToLive procedure....272
setReplyToDestinationType procedure....273
setReuseMessage procedure....274
setSaxReader procedure....274
setShort procedure....275
setShortProperty procedure....275
setShutdownWaitFor procedure....276
setString procedure....277
setStringProperty procedure....277
setTempTable procedure....278
setText procedure....278
setTransactedReceive procedure....279
setTransactedSend procedure....279
setUser procedure....280
setX-Document procedure....280
startReceiveMessages procedure....281
stopReceiveMessages procedure....282
subscribe procedure....282
waitForMessages procedure....283
writeBoolean procedure....284
writeByte procedure....285
writeBytesFromRaw procedure....286
writeChar procedure....286
writeDate procedure....287
writeDateTime procedure....288
writeDateTime-TZ procedure....289
writeDouble procedure....289

writeFloat procedure....290
writeInt procedure....290
writeInt64 procedure....291
writeLong procedure....292
writeLongString procedure....292
writeShort procedure....293
writeString procedure....294

Preface

Purpose

This content introduces the Progress OpenEdge JMS Adapter, the ABL-JMS APIs, general messaging concepts, and provides examples of implementing messaging.

Audience

This content is intended for OpenEdge administrators or ABL developers who need to create, configure, or implement messaging using the Progress OpenEdge JMS Adapter.

Organization

This content is organized in the following way:

- Learn about the Progress OpenEdge JMS Adapter on page 17

  Introduces the Progress OpenEdge JMS Adapter, prerequisites for working with it, and the main steps for using the adapter.

- Install the Progress OpenEdge JMS Adapter on page 19

  Explains how to install the Progress OpenEdge JMS Adapter using the installer or silently.

- Configure the Progress OpenEdge JMS Adapter on page 39

  Explains how to configure the Progress OpenEdge JMS Adapter to work with a third-party JMS provider, and start the adapter.

- Use the ABL-JMS Producer and Consumer APIs on page 49

  Describes how to use the ABL-JMS APIs with third-party messaging providers, and provides programming examples.

- Configure JMS-provider proprietary features on page 103

  Describes how to configure third-party JMS provider features.

- ABL-JMS API Reference on page 125

  Lists the ABL-JMS APIs in a reference format.

Documentation conventions

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

1

Learn about the Progress OpenEdge JMS Adapter

Progress OpenEdge provides a generic Java Message Service (JMS) adapter for implementing messaging in ABL applications. The Progress OpenEdge JMS Adapter can operate with any JMS 1.1-compliant vendor such as SonicMQ (now Aurea Messenger MQ), IBM WebSphereMQ, and ActiveMQ Artemis.

Note: The SonicMQ and Sonic ESB products were acquired by Aurea Software and became part of the Aurea CX Platform, which was subsequently renamed to Aurea Integration Solutions. In line with this, SonicMQ became Aurea CX Messenger MQ, and then was renamed to Aurea Messenger.

The Progress OpenEdge JMS Adapter is installed separately from the OpenEdge installation. The Progress OpenEdge JMS Adapter can be overlayed on an existing OpenEdge 12.2 installation, or configured separately as a standalone adapter. More information can be found in Install the Progress OpenEdge JMS Adapter on page 19

For details, see the following topics:

- Prerequisites for using the Progress OpenEdge JMS Adapter
- Main steps for using the Progress OpenEdge JMS Adapter

Prerequisites for using the Progress OpenEdge JMS Adapter

With the release of OpenEdge 12.2, Progress does not ship the SonicMQ client JAR files with the Progress OpenEdge JMS Adapter installer. If your organization wishes to continue using SonicMQ, it must obtain the client JAR files through Aurea MessengerMQ.

This content assumes that you have configured a JMS provider server and it is ready for use. Ensure that you have access to binaries that are required for the JMS provider.

You need to know the values of the following JMS provider properties:

- `PROVIDER_HOST:PROVIDER_PORT`: The hostname and port number of the JMS provider server.
- `PROVIDER_CONNECTION_FACTORY`: The name of the connection factory of the JMS provider.
- `PROVIDER_QUEUE, PROVIDER_TOPIC`: The names of the queues and topics of the JMS provider that are to be used.

Main steps for using the Progress OpenEdge JMS Adapter

The steps for using the Progress OpenEdge JMS Adapter are as follows:

1. Install the Progress OpenEdge JMS Adapter.
2. Configure the adapter to point to your third-party JMS provider.
3. Manage connections and sessions using JMS-provider proprietary features.
4. Use the ABL-JMS Producer and Consumer APIs.

2

Install the Progress OpenEdge JMS Adapter

To install the Progress OpenEdge Java Message Service (JMS) Adapter on Windows, download the OpenEdge JMS Adapter software image from the Progress Software Download Center, and launch the interactive installation program. The UNIX version of the OpenEdge JMS Adapter also offers an interactive installation utility. The software download is provided in a ZIP file. You need to extract its contents to your machine.

Note that during installation, you can optionally capture a `response.ini` file. You can use the `response.ini` file to create a silent installation script, which automates the installation of the OpenEdge JMS Adapter across a network.

Note: Download the `.jar` files from the respective JMS vendor that you are using. Export a class path if you are connecting to the JMS via a broker. You need to update the `javatools.properties` file if you are connecting via a client.

For details, see the following topics:

- Install the OpenEdge JMS Adapter interactively
- Install the OpenEdge JMS adapter on UNIX
- Install the OpenEdge JMS Adapter in silent mode

Install the OpenEdge JMS Adapter interactively

To launch the OpenEdge JMS Adapter, you must have administrator privileges on the machine where you are installing the OpenEdge JMS Adapter. For information about planning your installation, see 'Before you Install' in *Install OpenEdge*.

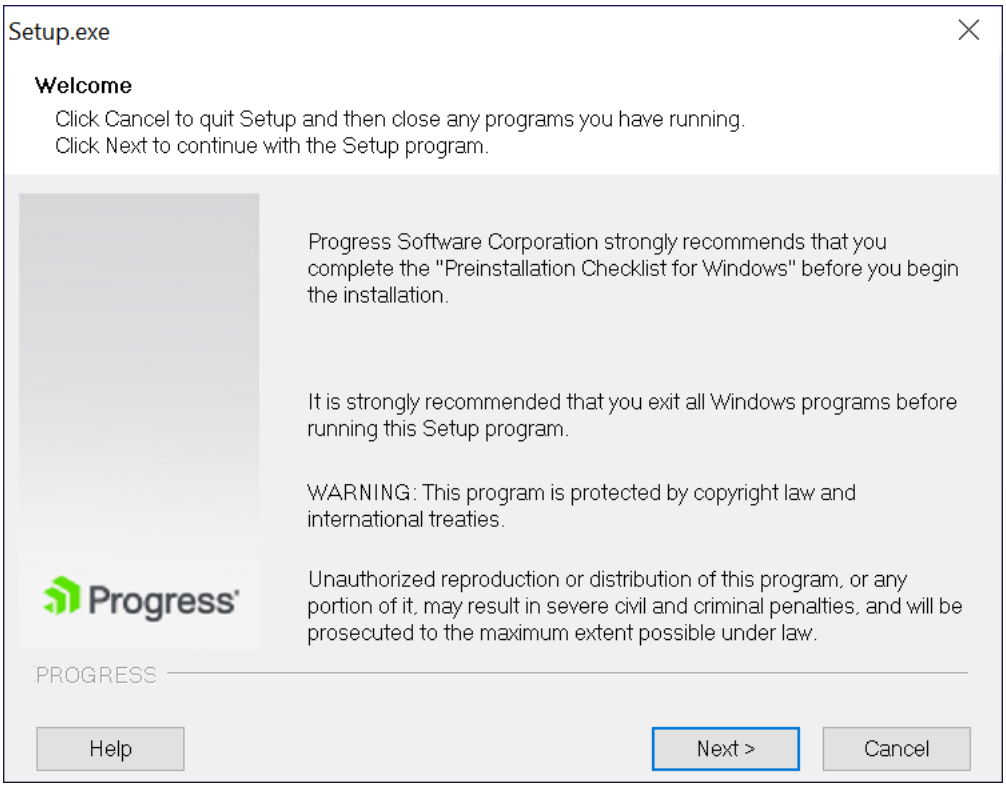
To install the OpenEdge JMS Adapter on your machine, complete the following steps:

1. Close all applications.

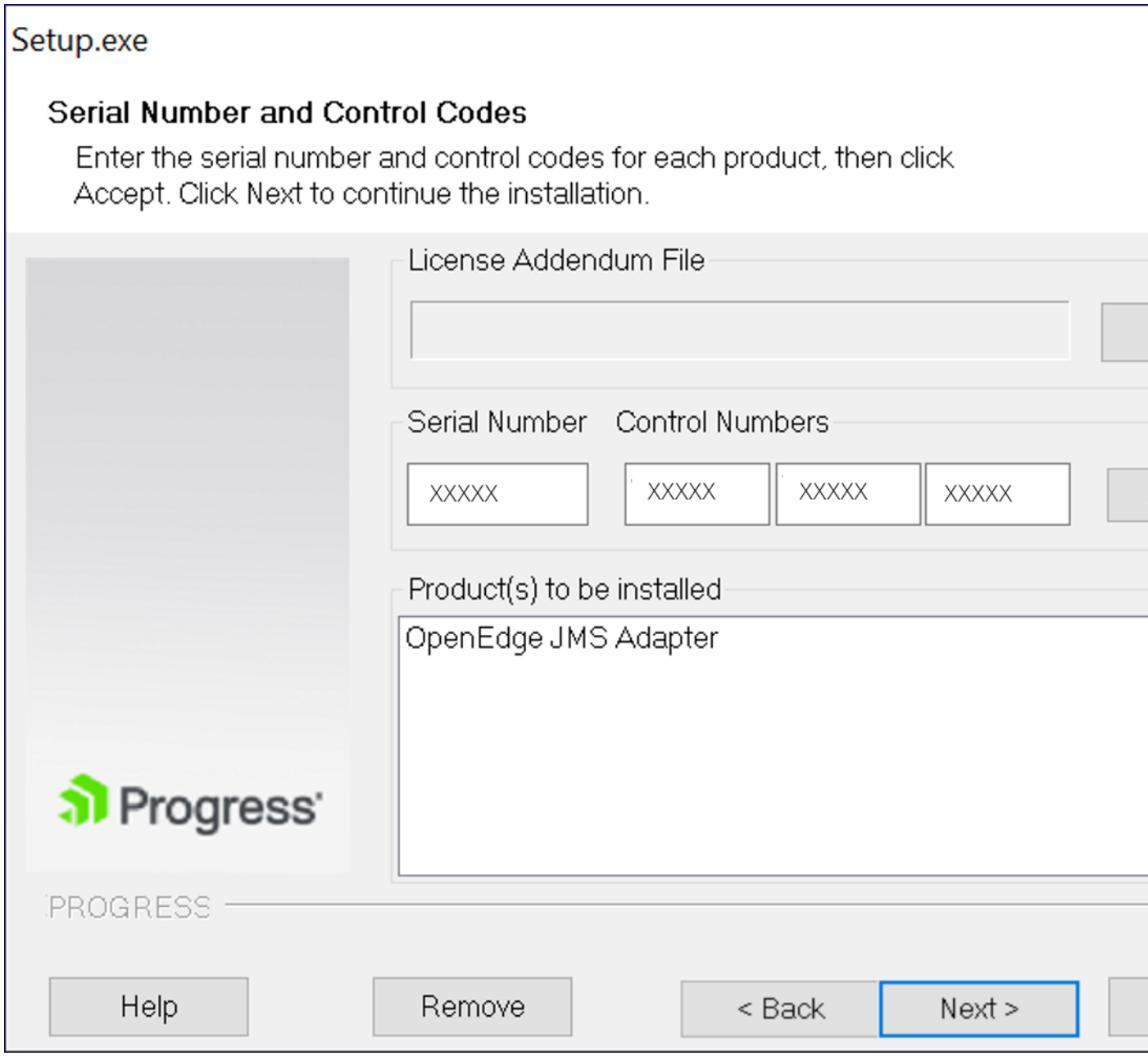
   Running applications or tasks may interfere with the installation because they could use files that OpenEdge needs to complete the installation. Stop any processes that are running, and close any files that are located in the same directory where you want to install the OpenEdge JMS Adapter.

2. Double-click the `setup.exe` file.

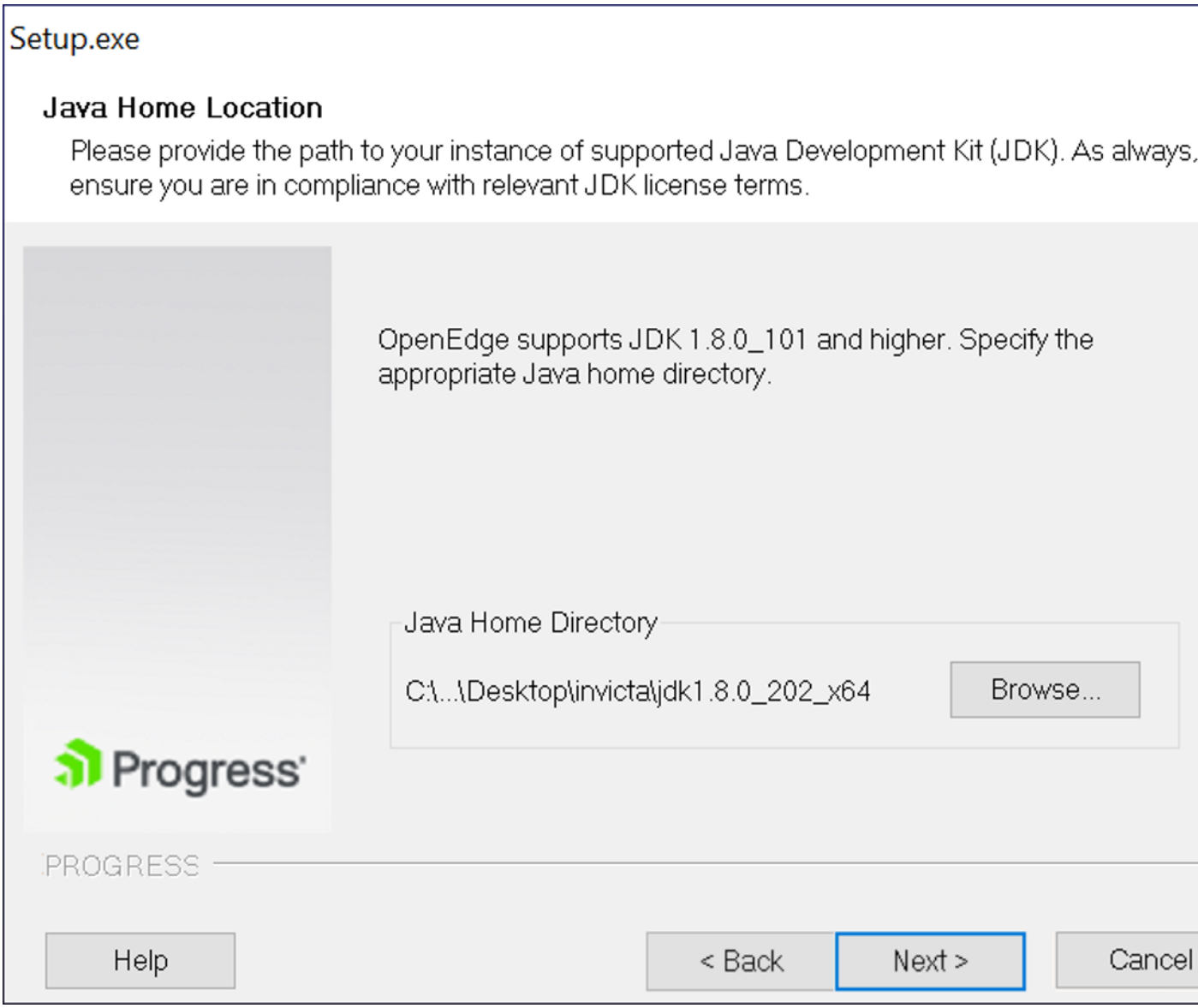
3. Read the information in the **Welcome** dialog box, verify that all the other applications are closed, and click **Next**.



4. In the **Serial Number and Control Codes** dialog box, enter the serial number and control number for the OpenEdge JMS Adapter and click **Accept**. Select **OpenEdge JMS Adapter** from the **Product(s) to be installed** box and click **Next**.


Setup.exe
Serial Number and Control Codes
Enter the serial number and control codes for each product, then click Accept. Click Next to continue the installation.
License Addendum File
Serial Number
Control Numbers
XXXXX
XXXXX
XXXXX
XXXXX
Product(s) to be installed
OpenEdge JMS Adapter
Progress
PROGRESS
Help
Remove
< Back
Next >

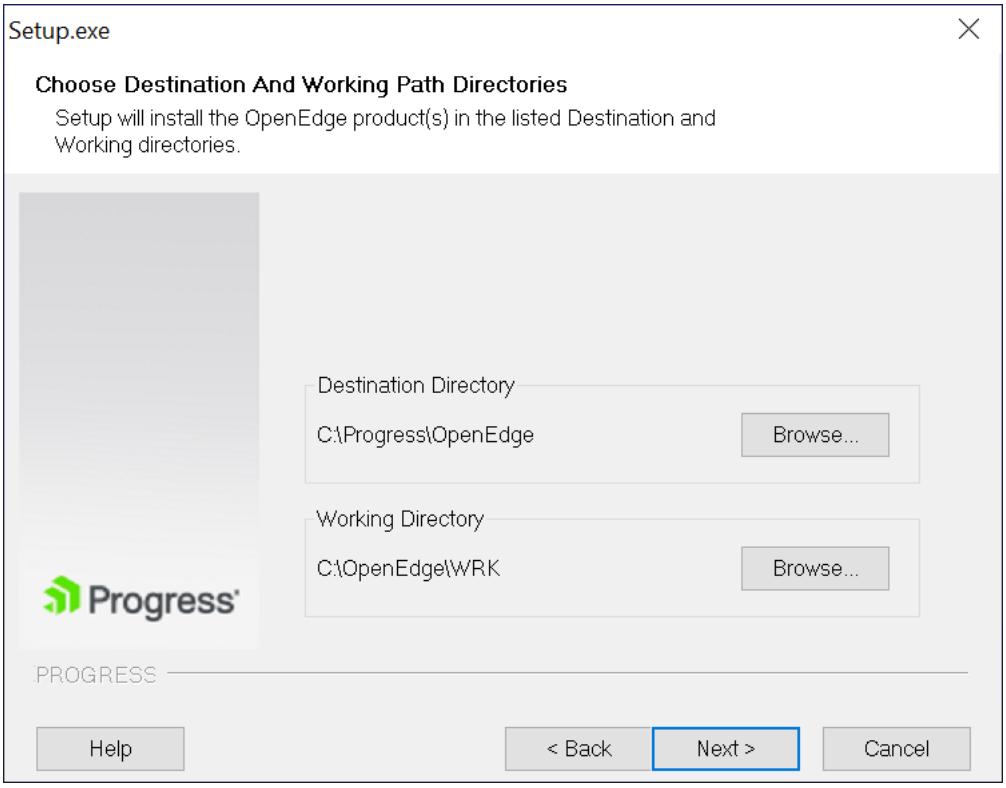
5. In the **Java Home Location** dialog box, click **Browse** and find the location of the JDK installation for the Java Home Directory. Click **Next**.



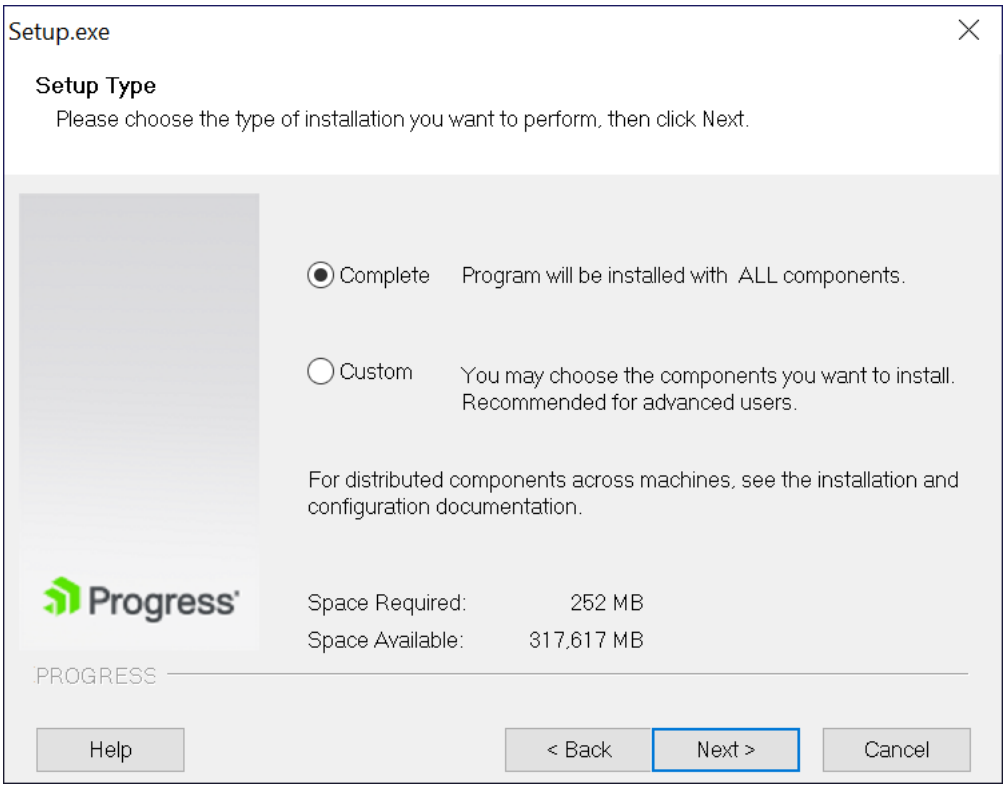
Note: Ensure that you have the correct JDK version installed. Otherwise, an error message is displayed, and you cannot proceed with the installation. For information about JDK requirements on each supported platform, see the OpenEdge 12.2 Platform and Product Availability Guide.

6. Specify the paths for the OpenEdge JMS Adapter destination and working directories in the **Choose Destination And Working Path Directories** dialog box. Click **Next**.

   The working directory contains your OpenEdge database and application files, and is the directory where OpenEdge runs. The working directory cannot be same as the destination directory.

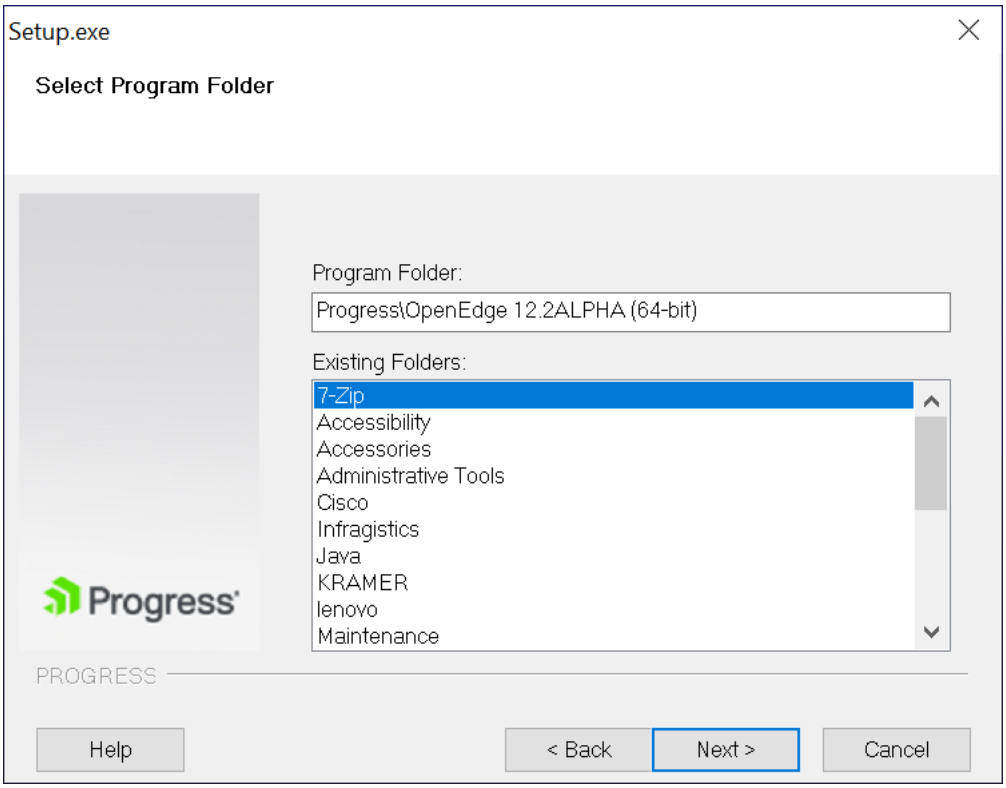


7. In the **Setup Type** dialog box, choose the type of installation you want to perform and click **Next**.

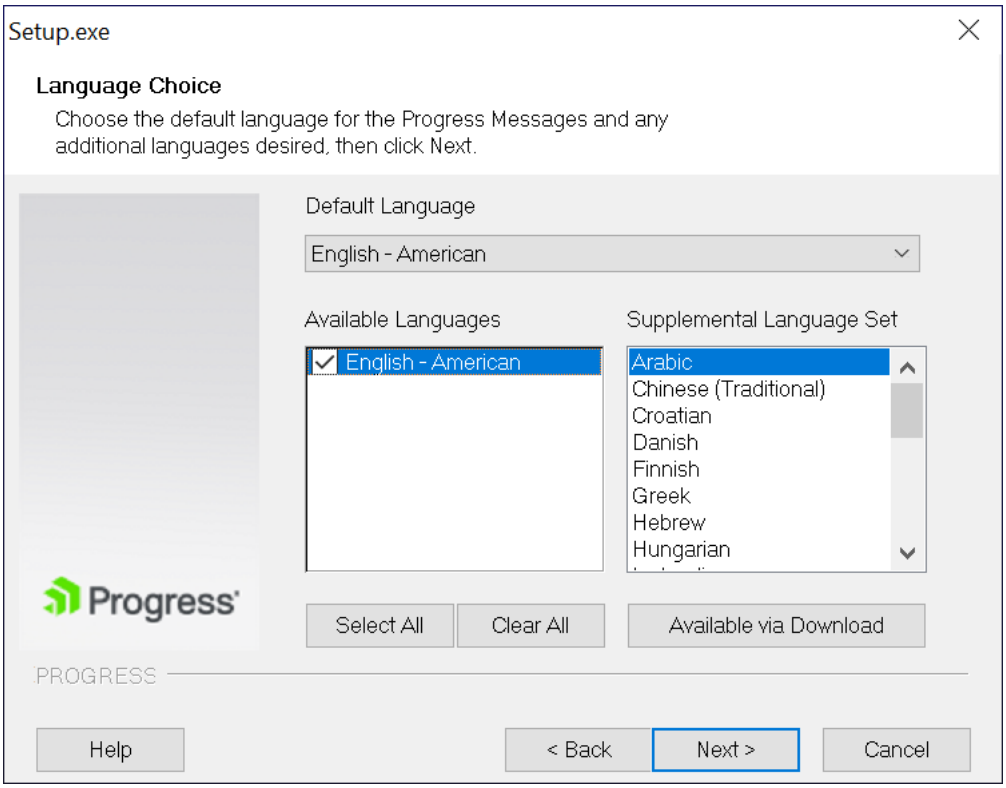


Note: For custom installations, Progress recommends that you install all recommended products. If you do not install all the recommended products, then the products you chose to install may not function as expected.

8. In the **Select Program Folder** dialog box, specify the Windows program folder where the OpenEdge JMS Adapter program icons will appear. Click **Next**.

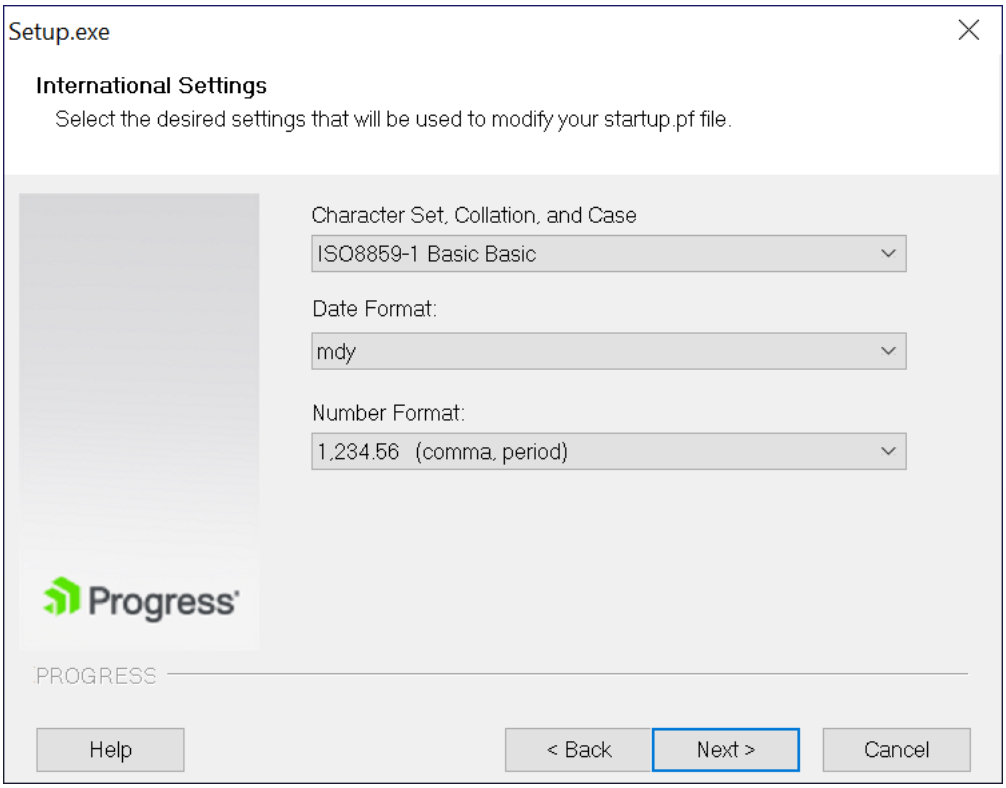


9. Choose the languages to be used for the OpenEdge JMS Adapter error and informational messages in the **Language Choice** dialog box. Click **Next**.



The Supplemental Language Set provides an additional list of languages in which messages can be displayed. If you select a supplemental language and choose **Available via Download**, an informational message appears, which indicates that you can obtain these supplemental languages from the Progress Download Center to **install** after you complete the **installation**.

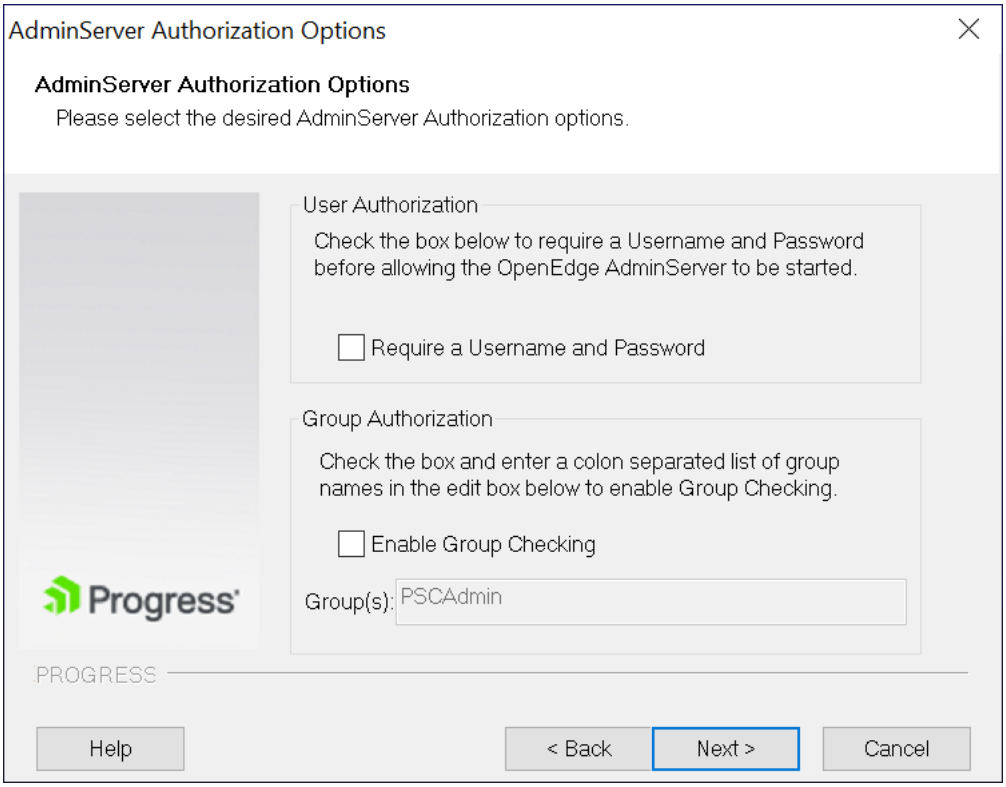
10. In the **International Settings** dialog box, choose the settings for your character set, date, and number format in OpenEdge software. Click **Next**.



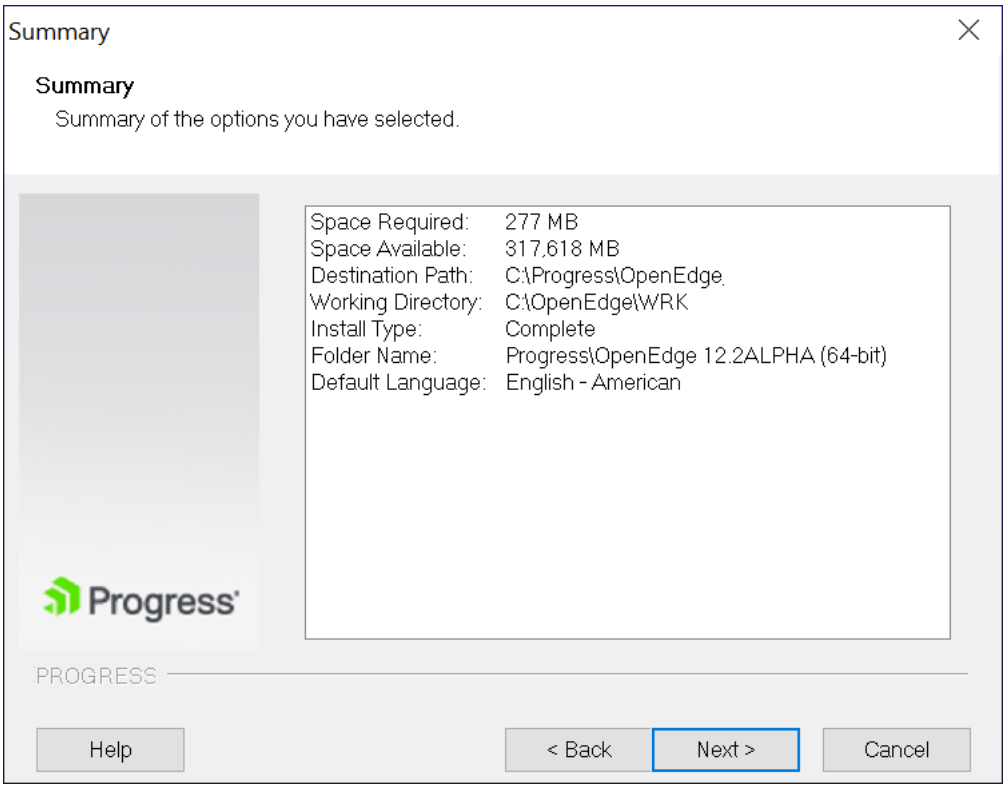
11. In the **AdminServer Authorization Options** dialog box, set up user and group authorization of the AdminServer for the OpenEdge JMS Adapter. You can choose one option, both options, or no options. After you make a choice, click **Next**.

    The **User Authorization** and **Group Authorization** options are established independently of each other.

    - For **User Authorization**, select **Require a Username and Password**. This option requires each user to provide a valid user name and password in order to start the AdminServer. User names and passwords are set up at the operating system level. For information about setting up users, see your operating system documentation.
    - For **Group Authorization**, select, **Enable Group Checking**. This option enables a specified group of users to be able to start the AdminServer.



12. In the **Summary** dialog box, review the installation options that you chose. Click **Back** to return to the previous dialog box to make any changes. If you are satisfied with your choices, then click **Next**.



13. When the installation is complete, the **Complete Setup Done** dialog box is displayed. Click **Finish** to exit the installation process.

Install the OpenEdge JMS adapter on UNIX

The OpenEdge JMS installation program on UNIX or Linux machines present a series of dialog boxes in which you enter your choices in each dialog box and advance at your own pace. The specific controls you use to advance to the next dialog box or return to a previous dialog box are identified on each dialog box. Select a menu option using the TAB key or the keys that are mentioned in each selection on the dialog box.

You can access the online help according to the method identified on each dialog box. Scroll in the help window to view all of the dialog box details. For information about planning your installation, see Before you Install in *Install OpenEdge*.

You can access the online help according to the method identified on each dialog box. Scroll in the help window to view all of the dialog box details. For information about planning your installation, see 'Before you Install' in *Install OpenEdge*.

To install the OpenEdge JMS adapter on your UNIX machine, complete the following steps:

1. Close all applications.

   Running applications or tasks may interfere with the installation because they could use files that OpenEdge needs to complete the installation. Stop any processes that are running, and close any files that are located in the same directory where you want to install the OpenEdge JMS adapter.

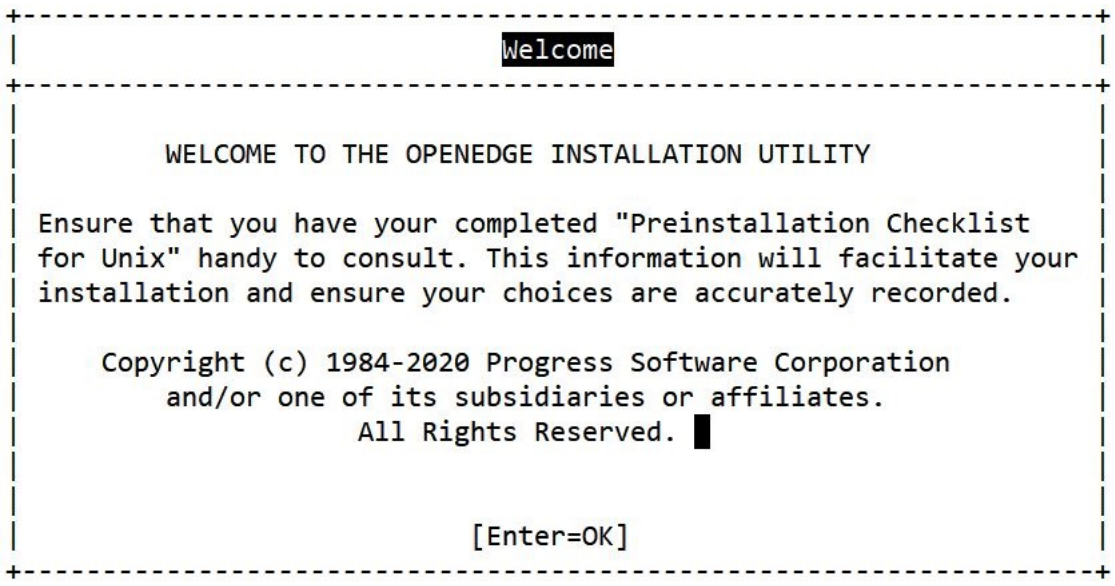
2. Log in as `root`. If you do not know the root password for your machine, check with your system administrator.

3. To start the installation, enter the following command:

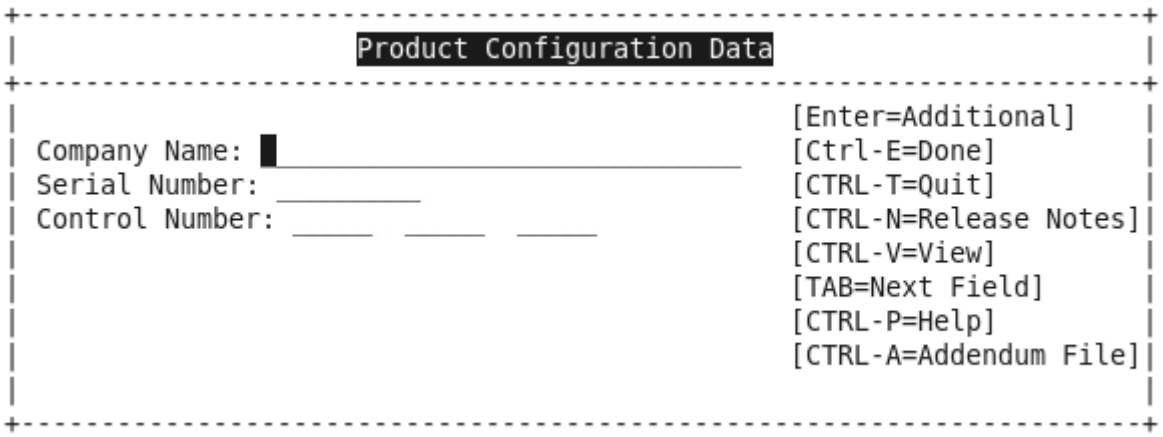
   ```
   ./path/proinst
   ```

   Note: In the preceding command, `path` represents the directory that contains the downloaded JMS adapter software image. You cannot run `proinst` from either the directory where you downloaded the OpenEdge JMS adapter software image or from the target directory in which you are installing the OpenEdge JMS adapter.

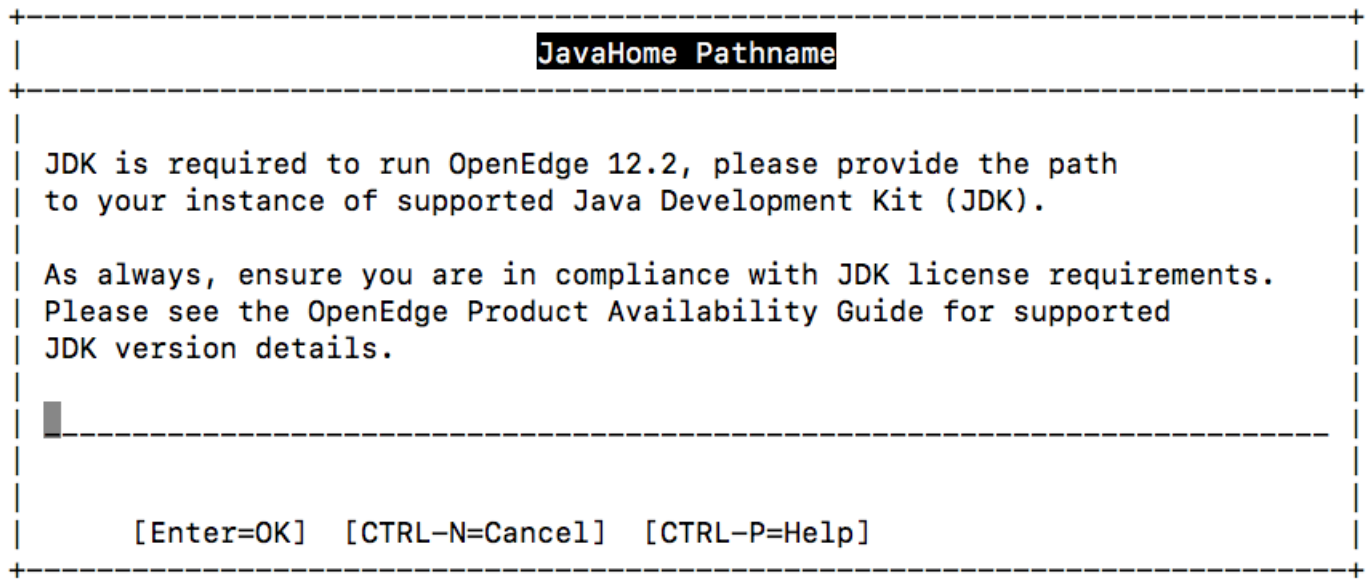
4. Read the information in the **Welcome** dialog box, verify that all the other applications are closed, and press ENTER.



5. In the **Product Configuration Data** dialog box, enter the company name, serial number, and control number for the OpenEdge JMS adapter. When you finish entering the license information, press CTRL + E to continue.

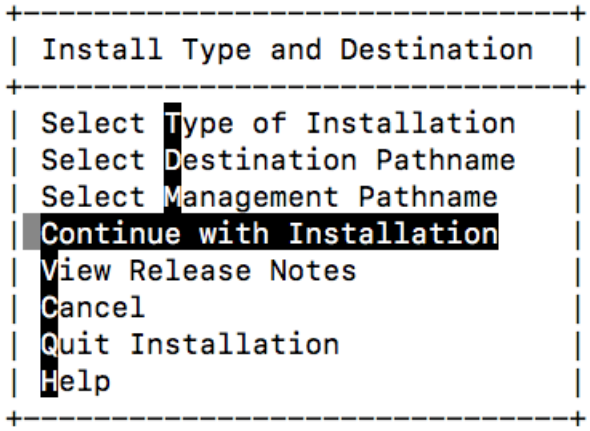


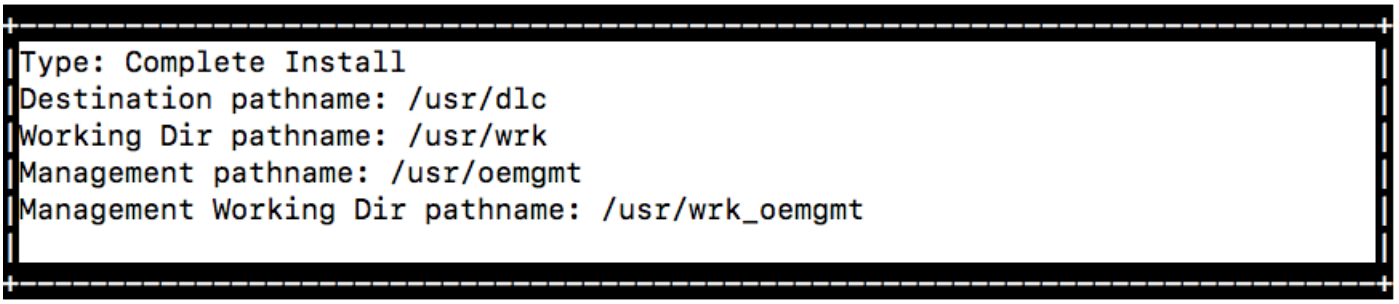
6. In the **JavaHome Pathname** dialog box, enter the the location of the JDK installation for the JavaHome and press ENTER.



Note: Ensure that you have the correct JDK version installed. Otherwise, an error message is displayed, and you cannot proceed with the installation. For information about the JDK requirements on each supported platform, see the OpenEdge 12.2 Platform and Product Availability Guide.

7. In the **Install Type and Destination** dialog box, select one of the options, press TAB to highlight `Continue with Installation`, and then press ENTER.



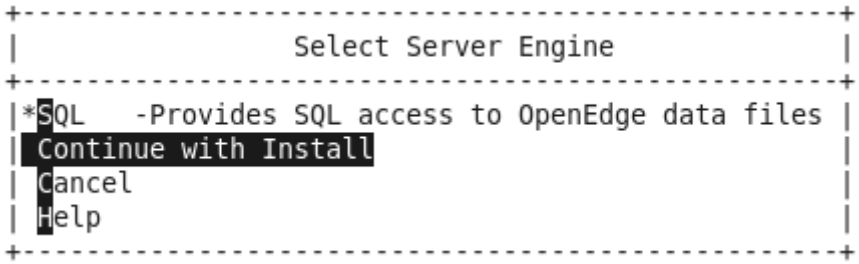


For custom installations, Progress recommends that you install all recommended products. If you do not install all the recommended products, then the products you chose to install may not function as expected.

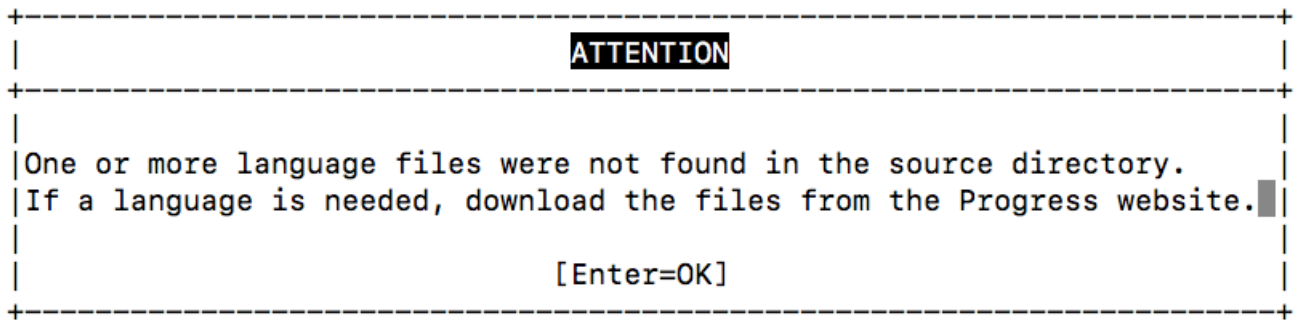
Specify the paths for the OpenEdge JMS adapter destination and working directories in the `Destination pathname` and `Working Dir pathname`,respectively.

The working directory contains your OpenEdge JMS adapter database and application files, and is the directory where the OpenEdge JMS adapter runs. The working directory cannot be same as the destination directory.

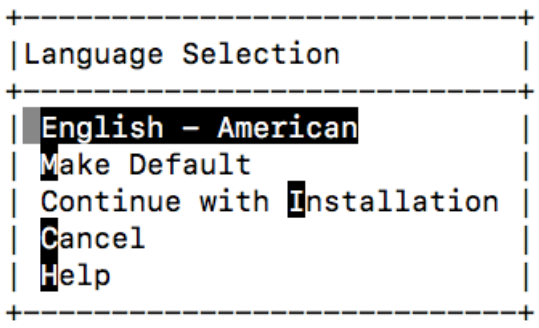
If the installation program prompts you to specify whether you want to select the SQL Server database engine to support queries written in SQL, then press the TAB key to highlight `Continue with Install`, and then press ENTER.



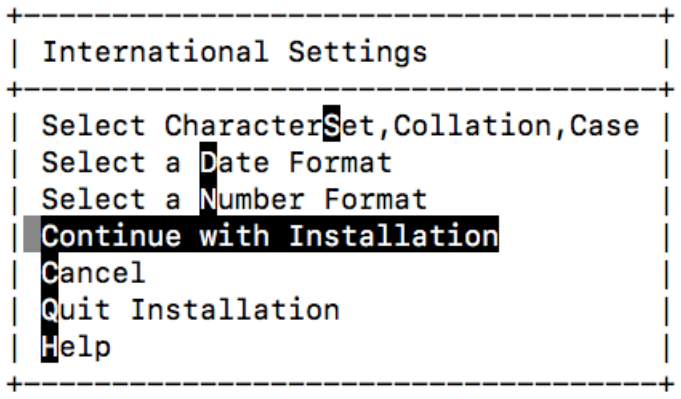
8. In the **Attention** dialog box, press ENTER.

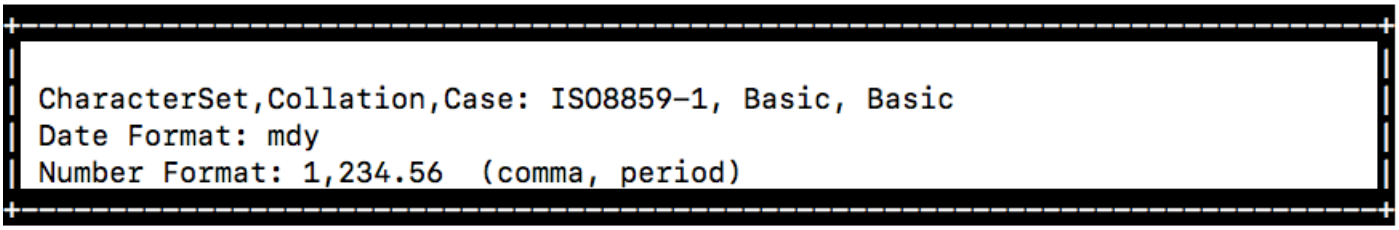


9. In the **Language Selection** dialog box, select `English - American` as the language to be used for OpenEdge error and informational messages. Press the TAB key to highlight `Make Default`, and then press ENTER. Now, press the TAB key to highlight `Continue with Installation`, and then press ENTER.

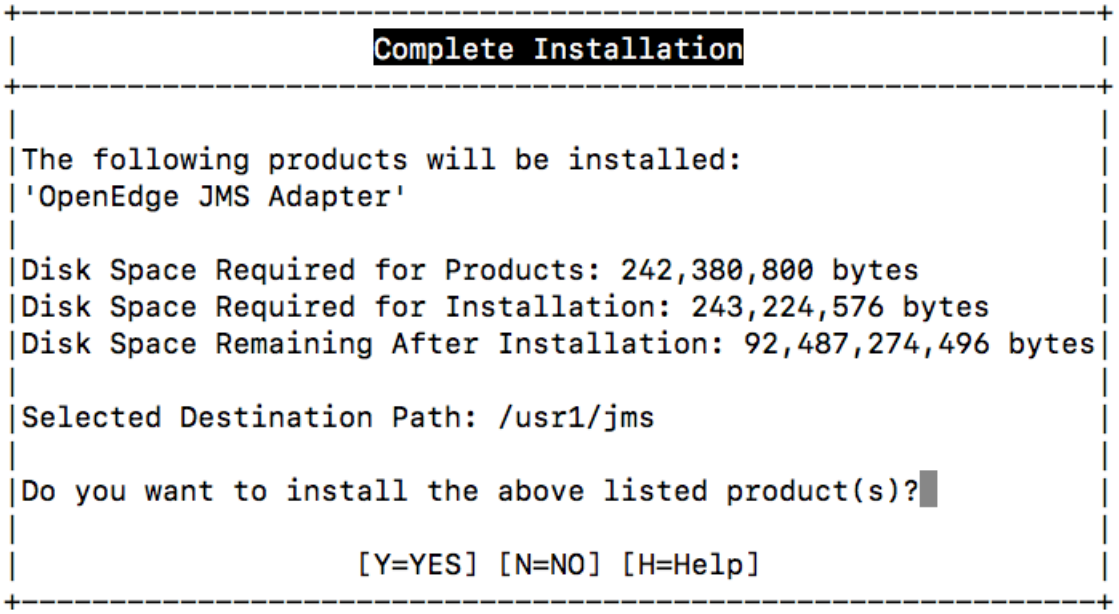


10. In the **International Settings** dialog box, choose the settings for your character set, date, and number format in OpenEdge software. Details about the information required for this dialog box are available from online help. Press the TAB key to highlight `Continue with Installation`, and then press ENTER.

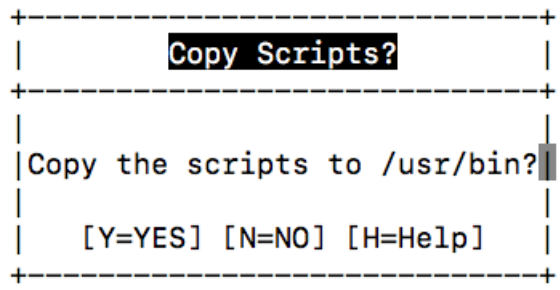




11. In the **Complete Installation** dialog box, review the installation options that you chose. Press N to return to the previous dialog box to make any changes. If you are satisfied with your choices, then press Y.

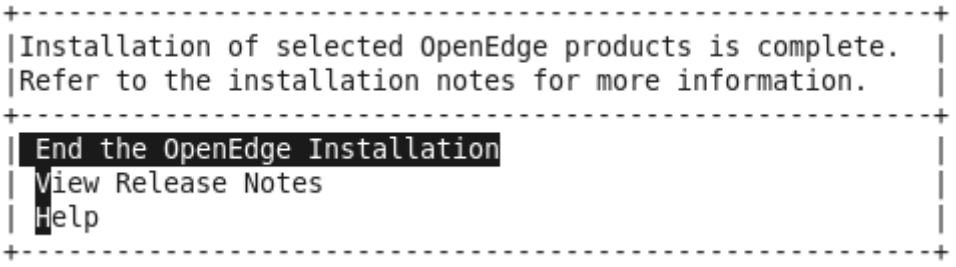


12. In the **Copy Scripts** dialog box, confirm whether you want to copy the OpenEdge JMS adapter product scripts and program modules to the default location `/usr/bin`. Press Y or N to continue.



Note: If you are maintaining two versions of OpenEdge on the same machine, press `N` to avoid overwriting files.

13. When the installation is complete, a message is displayed. Press the TAB key to highlight the `End the OpenEdge Installation`, and press ENTER to exit the installation process.



Install the OpenEdge JMS Adapter in silent mode

An interactive installation prompts you for information, and records the values you chose using a series of dialog boxes. When you start an interactive installation, the data that you enter is recorded in a response file. On UNIX and Linux machines, the default name for the response file is `linuxx86_64.ini`. On Windows machines, the default name for the response file is `response.ini`, which is created in the install subdirectory of your OpenEdge JMS Adapter installation directory.

For more information about response files, see 'Contents of the response file' in *Install OpenEdge*.

A silent installation performs an installation using a script, and is a two-step process:

1. Perform an interactive installation.
2. Execute the setup command and use the response file to provide values for the installation.

The installation data that is captured in a response file can be played back, to perform a silent installation through a batch mechanism. You can modify the contents of the response file that is generated by the installation program, or you can create a response file. However, for a silent installation, Progress recommends that you use the automatically generated and unmodified response file.

Note: A supported Java SE Development Kit (JDK) must be installed on your machine before running the OpenEdge Java Message Servie (JMS) Adapter silent installation program. The JDK is not included with the OpenEdge JMS Adapter software. Ensure that you edit the location of the directory in which the JDK is installed in the response file before the installation procedure. For information about JDK requirements, see the OpenEdge 12.2 Platform and Product Availability Guide.

The command that you use to initiate or playback the response file is the same, regardless of which data input option you choose. The OpenEdge JMS Adapter silent installer runs without intervention after you enter the command to start the process.

Syntax to install the OpenEdge JMS adapter on Windows

```
path-to-install-media\setup.exe -psc_s [-notify]
-psc_f1=path\response-file [-psc_f2=path\logfile]
```

In the preceding syntax:

- `path-to-install-media\setup.exe` runs the OpenEdge JMS Adapter installation. You can run the installation from the directory that contains the installation executable that you downloaded from the Progress Download Center.
- `-psc_s` specifies a silent installation.
- `-notify` specifies that the installation dialog boxes that are displayed contain details about the current installation phase and the completion percentage.

  Progress recommends that you set up your installation to poll the log file for the status of the installation. See 'Check the status of the silent installation log file' in *Install OpenEdge* to review the contents of the runtime status and result code information in the sample `oesetup.log` file.
- `-psc_f1=path\response-file` specifies the full path of the response file. By default, the silent installation searches for the response file `oesetup.ini` in the same directory where `setup.exe` is located.
- `-psc_f2=path\logfile` creates an installation log file. If no path or name is specified, then the file `oesetup.log` is created in the Windows directory by default.

Example

The following command executes a silent installation on the Windows platform:

```
\\cd-server\OpenEdge\setup.exe
-psc_s -psc_f1=C:\SilentInstalls\response.ini
    -psc_f2=C:\SilentInstalls\oesetup.log
```

Syntax to install the OpenEdge JMS adapter on UNIX

```
proinst -b path/install-ini-name -l
path/logfile [-n]
```

In the preceding syntax:

- `proinst` is the command that initiates the OpenEdge JMS adapter installation..
- `-b` *`path/install-ini-name`* performs a batch installation, specifying the full path of the `.ini` file to be used for the silent installation. You can use the `linuxx86_64.ini` file or another `.ini` file that you create.
- `-l` *`path/logfile`* creates a log file, optionally specifying the full path of the log file in which the installation events are logged. By default, the log file `install.log` is created in the directory identified by the first environment variable that the installer finds among the following: `$TMP`, `$TEMP`, or `$TMPDIR`.
- `-n` specifies that the batch installation includes a progress indicator, displaying details about the current installation phase and completion status.

Example

The following command executes a silent installation on the UNIX platform:

```
proinst -b /test/linuxx86_64.ini -l /log/test.log
```

3

Configure the Progress OpenEdge JMS Adapter

To use the Progress OpenEdge JMS Adapter, you must point OpenEdge to the JMS provider that you are using. To achieve this, OpenEdge has to find the connection factory specific to your JMS provider. These topics details how to create the connection between OpenEdge and your JMS provider.

Note: The Progress OpenEdge JMS Adapter supports the ClientConnect and BrokerConnect modes. For more information, see Select the JMS Adapter connection mode on page 50.

For details, see the following topics:

- Configure your JMS provider
- Use the Progress OpenEdge JMS Adapter and the ABL-JMS API with JMS Administered Objects
- Build the AdminObjectFinder for use with JNDI
- Start the Progress OpenEdge JMS Adapter
- Configure logging for the Progress OpenEdge JMS adapter

Configure your JMS provider

The prerequisites for this content assume that you have configured a JMS provider server and it is ready for use, and the Progress OpenEdge JMS Adapter is installed.

Ensure that you have access to binaries that are required for the JMS provider, and then follow these steps to configure your JMS provider to work with the Progress OpenEdge JMS Adapter:

1. Add the JMS provider client JAR files to the classpath in the `%DLC%\properties\JavaTools.properties` file under the `[SonicMQ]` property section, as in the following examples:

```
[SonicMQ]
classpath=C:\JMS\IBM\WEBSPHERE\java\lib\com.ibm.mqjms.jar,${DLC}/java/progress.jar
```

```
[SonicMQ]
classpath=C:\JMS\ArtemisMQ\apache-artemis-2.11.0-bin\apache-artemis-2.11.0\lib\client\artemis-jms-client-all-2.11.0.jar,${DLC}/java/progress.jar
```

Note: SonicMQ client JAR files are not shipped with OpenEdge 12.2. If you are using SonicMQ as a JMS provider, you must also add them to the classpath. The default is `$DLC/sonic/MQ10.0/lib/sonic_client.jar`. For example:

```
[SonicMQ]
classpath=$DLC/sonic/MQ10.0/lib/sonic_client.jar,${DLC}/java/progress.jar
```

If there is more than one jar file, provide the paths as semi-colon separated values.

2. Update the `jvmargs` and `DjmsProvider` properties in the `%DLC%\properties\JavaTools.properties` file in the `[SonicMQ]` property section to specify your JMS provider.

   If you are using SonicMQ as a JMS provider, add the following entry to the `[SonicMQ]` property section below the `classpath`:

```
jvmargs=-DsonicMQExtensions=true -DjmsProvider=SonicMQ
```

   This provides access to the SonicMQ proprietary APIs.

   If you are configuring another third-party JMS provider, add the following entry to the `[SonicMQ]` property section below the `classpath`:

```
jvmargs=-DsonicMQExtensions=false -DjmsProvider=ProviderName
```

   Where *`ProviderName`* is the name of your JMS provider, such as WebSphereMQ or ArtemisMQ. The *`ProviderName`* must match the JMS provider specified in the `jmsProvider.properties` file. Third-party JMS providers outside of SonicMQ can use JMS Administered Objects to access the provider-specific proprietary APIs.

3. Specify the JMS Provider and define the Connection Factory in the `%DLC%\properties\jmsProvider.properties` file. There are several third-party JMS providers that are pre-defined that provide Connection Factory class names. If your JMS provider is not listed, then add it to the list with the following format:

```
[MyJMSProvider]
javax.jms.ConnectionFactory=
javax.jms.QueueConnectionFactory=
javax.jms.TopicConnectionFactory=
```

Note: Do not modify the `javax.jms.*` property names. The property values for these property names must be specific to the JMS provider.

4. If you require an update to your JDK version or Java Path, you can update the file location that you specified in the Progress OpenEdge JMS Adapter installation using the `%DLC%\properties\JMSAdapterJava.properties` file.

Use the Progress OpenEdge JMS Adapter and the ABL-JMS API with JMS Administered Objects

JMS does not impose any specific directory for storing administered objects (although it establishes the convention of using JNDI-compliant directories, such as LDAP). Also, the process of connecting to a JNDI server and obtaining an initial context is not standardized.

Therefore, to use directory-stored JMS objects, you must implement a Java class, compile it, and install the class file on the JavaTools.properties file on the Progress OpenEdge JMS Adapter for [SonicMQ] `CLASSPATH` (regardless of JMS-provider). The Progress OpenEdge JMS Adapter looks for that class when it starts up. If it finds the class, it creates an instance object of it and uses it to locate administered objects. If it does not find the class, the Progress OpenEdge JMS Adapter creates objects as required.

For more information, see the next topic: Build the AdminObjectFinder for use with JNDI on page 42.

Build the AdminObjectFinder for use with JNDI

You can locate the connection factory by using the Java Naming and Directory Interface (JNDI) to find the JMS-administered objects in the JNDI namespace.

To prevent class conflict with the `AdminObjectFinder.java` file, the Progress OpenEdge JMS Adapter packages the class in a different package, `jmsfromABL` (not `jmsfrom4gl`). This new class provides the adapter with the configuration to find the JMS-administered objects in the JNDI namespace.

1. Create the `jmsfromABL.AdminObjectFinder.java` class file for your JMS provider. For example code, see the following topics:

   - AdminObjectFinder.java example for IBM WebsphereMQ on page 43
   - AdminObjectFinder.java and jndi.properties example for ActiveMQ Artemis on page 45

Note:

- The `jmsfromABL.AdminObjectFinder` name is mandatory in the class file.
- The class and the get...() methods must be declared as public elements.

2. Compile the `jmsfromABL.AdminObjectFinder.java` class file with the JMS-provider client jars with JDK 8 and place it in a new JAR file. For example:

```
C:\JMS\SonicMQ>copy C:\JMS\SonicMQ\MQ10.0\lib\sonic_Client.jar .
C:\JMS\SonicMQ>javac -cp sonic_Client.jar;.; -d . AdminObjectFinder.java
C:\JMS\SonicMQ>jar -cvf AdminObjectFinder.jar jmsfromABL
```

For ActiveMQ Artemis, also include the `jndi.properties` file in the JAR file. For example:

```
C:\JMS\ArtemisMQ>copy
C:\JMS\ArtemisMQ\apache-artemis-2.11.0-bin\apache-artemis-2.11.0\lib\client\artemis-jms-client-all-2.11.0.jar
 .
C:\JMS\ArtemisMQ>javac -cp artemis-jms-client-all-2.11.0.jar;.; -d .
AdminObjectFinder.java
C:\JMS\ArtemisMQ>jar -cvf AdminObjectFinder.jar jmsfromABL jndi.properties
```

3. Add the location of this new JAR file to the `classpath` variable in the `JavaTools.properties` file of the `[SonicMQ]` property section, as in the following example:

```
[SonicMQ]
classpath=C:\JMS\ArtemisMQ\artemis-jms-client-all-2.11.0.jar,${DLC}/java/progress.jar,C:\JMS\ArtemisMQ\AdminObjectFinder.jar
jvmargs=-DsonicMQExtensions=false -DjmsProvider=ArtemisMQ
```

AdminObjectFinder.java example for IBM WebsphereMQ

Use the following code to create an `AdminObjectFinder.java` file IBM WebsphereMQ:

```
package jmsfromABL;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.TopicConnectionFactory;
import javax.jms.QueueConnectionFactory;
import javax.jms.Topic;
import javax.jms.Queue;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NameNotFoundException;
import javax.naming.NamingException;
import java.util.Hashtable;

public class AdminObjectFinder{

    public Context context = null;
    public AdminObjectFinder() throws Exception{

        Hashtable<String, String> env = new Hashtable<>();
```

```
        env.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.sun.jndi.fscontext.RefFSContextFactory");
        env.put(Context.PROVIDER_URL, "file:/C:/JNDI");
        env.put(Context.SECURITY_PRINCIPAL, "username");
        env.put(Context.SECURITY_CREDENTIALS, "password");
        context = new InitialContext(env);
    }

    public TopicConnectionFactory getTopicConnectionFactory(String name)
    throws Exception {
        TopicConnectionFactory factory = null;
        factory = (javax.jms.TopicConnectionFactory)context.lookup(name);
        return factory;
    }

    public QueueConnectionFactory getQueueConnectionFactory(String name)
    throws Exception {
        QueueConnectionFactory factory = null;
        factory = (javax.jms.QueueConnectionFactory)context.lookup(name);
        return factory;
    }

    public ConnectionFactory getConnectionFactory(String name)
    throws NamingException {
        return (ConnectionFactory) context.lookup(name);
    }

    public Topic getTopic(String name) throws Exception {
        Topic topic = null;
        Object object = null;
        object = context.lookup(name);
        if (object != null) {
            topic = (javax.jms.Topic) object;
        }
        return topic;
    }

    public Queue getQueue(String name) throws Exception {
        Queue queue = null;
        Object object = null;
        object = context.lookup(name);
        if (object != null) {
            queue = (javax.jms.Queue) object;
        }
        return queue;
    }

    public Destination getDestination(String name)
    throws NamingException {
        return (Destination) context.lookup(name);
    }
}
```

Where:

`env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory")`

Specifies the name of the context factory class for WebSphereMQ.

Note: This example assumes that JNDI .bindings file is placed at a physical location in the FileSystem and thus is referred to as RefFileSystemContextFactory. This .binding file contains attributes that are used to create a ConnectionFactory object and other JNDI resources. For WebSphereMQ, the attributes to create a connection factory resides in a .bindings file.

```
env.put(Context.PROVIDER_URL, "file:/C:/JNDI")
```

Specifies the location of the JNDI .bindings file. For WebSphereMQ, the location of .bindings file.

```
env.put(Context.SECURITY_PRINCIPAL, "username");
env.put(Context.SECURITY_CREDENTIALS, "password")
```

Specifies the security credentials to access the MQService.

Note:

- The jmsfromABL.AdminObjectFinder name is mandatory in the class file.
- The class and the get...() methods must be declared as a public element.

AdminObjectFinder.java and jndi.properties example for ActiveMQ Artemis

Configuring JNDI for ActiveMQ Artemis requires and `AdminObjectFinder.java` and a `jndi.properties` file. The `jndi.properties` file specifies the attributes to create a connection factory.

Use the following code to create an `AdminObjectFinder.java` file for ActiveMQ Artemis:

```
package jmsfromABL;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.TopicConnectionFactory;
import javax.jms.QueueConnectionFactory;
import javax.jms.Topic;
import javax.jms.Queue;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NameNotFoundException;
import javax.naming.NamingException;
import java.util.Hashtable;
import java.io.File;

public class AdminObjectFinder{

    public Context context = null;
    public AdminObjectFinder() throws Exception{
        context = new InitialContext();
    }

    public TopicConnectionFactory getTopicConnectionFactory(String name)
    throws Exception {
        TopicConnectionFactory factory = null;
        factory = (javax.jms.TopicConnectionFactory)context.lookup(name);
        return factory;
    }

    public QueueConnectionFactory getQueueConnectionFactory(String name)
    throws Exception {
        QueueConnectionFactory factory = null;
        factory = (javax.jms.QueueConnectionFactory)context.lookup(name);
        return factory;
    }

    public ConnectionFactory getConnectionFactory(String name)
    throws NamingException {
```

```
        return (ConnectionFactory) context.lookup(name);
    }

    public Topic getTopic(String name) throws Exception {
        Topic topic = null;
        Object object = null;
        object = context.lookup(name);
        if (object != null) {
            topic = (javax.jms.Topic) object;
        }
        return topic;
    }

    public Queue getQueue(String name) throws Exception {
        Queue queue = null;
        Object object = null;
        object = context.lookup(name);
        if (object != null) {
            queue = (javax.jms.Queue) object;
        }
        return queue;
    }

    public Destination getDestination(String name)
    throws NamingException {
        return (Destination) context.lookup(name);
    }
}
```

Use the following code to create a `jndi.properties` file for ActiveMQ Artemis:

```
# START SNIPPET: jndi

java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory

# use the following property to configure the default connector
java.naming.provider.url=failover:(tcp://localhost:61617,tcp://localhost:61618)?initialReconnectDelay=100
#java.naming.provider.url=tcp://localhost:61617

# use the following property to specify the JNDI name the connection factory
# should appear as.
connectionFactoryNames = ConnectionFactory

# register some queues in JNDI using the form
# queue.[jndiName] = [physicalName]
queue.MyQueue = example.MyQueue


# register some topics in JNDI using the form
# topic.[jndiName] = [physicalName]
topic.MyTopic = example.MyTopic

# END SNIPPET: jndi
```

Note: Replace the hostname and port in the above example code with your failover broker's location.

Start the Progress OpenEdge JMS Adapter

To start the Progress OpenEdge JMS Adapter, follow these steps:

1. Using proenv, set the JMSPROVIDER environment variable. For example:

```
proenv>set JMSPROVIDER=ProviderName
```

   where *ProviderName* is the JMS provider that is specified in the `-DjmsProvider` parameter in `JavaTools.properties`.

2. Using proenv, Set the JMSCLIENTJAR environment variable. For example:

```
proenv>set JMSCLIENTJAR=JMSCLIENTJAR_path
```

   where *JMSCLIENTJAR_path* is the full path of the JMS client JAR file. The default is `$DLC/sonic/MQ10.0/lib/sonic_client.jar`. If there are more than one jar file, provide the paths as semi-colon separated values.

3. Start the Progress OpenEdge JMS Adapter broker using the `oemessaging start` command. For example:

```
proenv>oemessaging start
```

4. After you have started the OpenEdge JMS Adapter, all available administrative options with their short description are listed in the proenv window. The options are as follows:

   - **S** — Summary of the JMS Adapter broker
   - **D** — Server Details
   - **X** — Add new agents
   - **T** — Trim free agents
   - **K** — Kill by server ID
   - **E** — Stop and Exit the broker gracefully
   - **A** — Abort
   - **L** — Connection Summary
   - **C** — Connection detail for one or more connection handles
   - **Y** — List JMS Adapter properties
   - **Z** — List broker Property value based on property name

Configure logging for the Progress OpenEdge JMS adapter

To configure logging for ClientConnect mode of the Progress OpenEdge JMS Adapter, edit the `$DLC/properties/javaTools.properties` file to increase the logging level. For example, find and edit these properties:

```
[Adapter.CC.cc1]
               brkrLoggingLevel=3
               srvrLoggingLevel=3
```

This produces `cc.broker` and `cc.server` files in the directory where the client procedures were run.

To change broker and server logging levels for BrokerConnect mode, edit the `$DLC/properties/uBroker.properties`. For example, find and edit these properties:

```
[Adapter.genericMQ1]
    brokerLogFile=@{WorkPath}\genericMQ1.broker.log
    srvrLogFile=@{WorkPath}\genericMQ1.server.log

[UBroker]
    brkrLogEntryTypes=UBroker.Basic
    brkrLoggingLevel=2
    srvrLogEntryTypes=
    srvrLoggingLevel=2
```

Logging is turned on by setting the `brkrLoggingLevel` and `srvrLoggingLevel` properties to a value greater than 0, and setting `brkrLogEntryTypes=UBroker.Basic` (the default).

Note: Logging level represents the amount of logging information written to the log file. Valid logging level values are 1 (Errors), 2 (Basic), 3 (Verbose), and 4 (Extended). The default value is 2. For more information on logging, see *Troubleshoot ABL Applications*.

4

Use the ABL-JMS Producer and Consumer APIs

In order to exchange messages using JMS messaging model, an OpenEdge client establishes a connection to a SonicMQ Broker, creates the message, and sends the message to a receiver.

This chapter details how to Select the JMS Adapter connection mode on page 50, and how to write and consume messages.

Programming examples are provided in:

- PTP message example on page 82
- Pub/Sub messaging example on page 85
- Programming scenarios on page 88

For an alphabetical API reference, see ABL-JMS API Reference on page 125.

For details, see the following topics:

- Select the JMS Adapter connection mode
- Manage connections and sessions
- Work with messages
- Consume messages
- Transacted sessions
- Message acknowledgement, forwarding, and recovery
- Error and condition handling

- PTP message example
- Pub/Sub messaging example
- Programming scenarios

Select the JMS Adapter connection mode

The Progress OpenEdge JMS Adapter supports the following two connection modes:

- Progress OpenEdge JMS Adapter BrokerConnect
- Progress OpenEdge JMS Adapter ClientConnect

If the Progress OpenEdge JMS Adapter is installed using the standalone installer without being overlayed on an OpenEdge 12.2 installation, only the BrokerConnect mode is supported.

If the Progress OpenEdge JMS Adapter is installed using the standalone installer but overlayed on an existing OpenEdge 12.2 installation using the same $DLC location, the ClientConnect and BrokerConnect modes are supported.

See also

Progress OpenEdge JMS Adapter BrokerConnect on page 50
Progress OpenEdge JMS Adapter ClientConnect on page 52

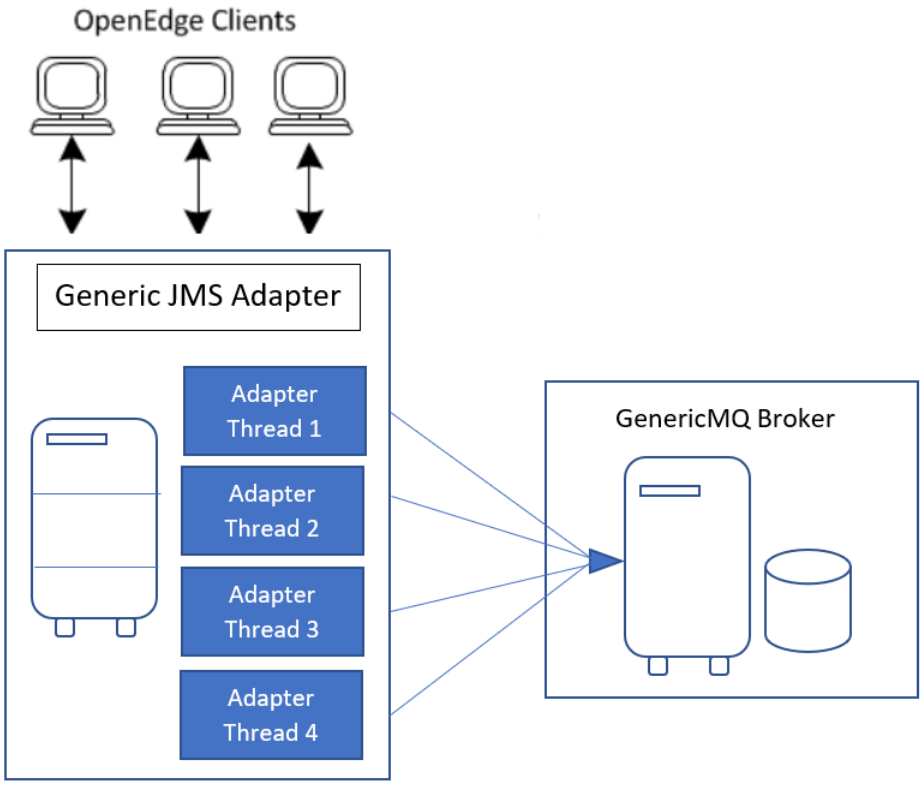
Progress OpenEdge JMS Adapter BrokerConnect

The Progress OpenEdge JMS Adapter allows you to connect to third-party JMS providers including SonicMQ, ActiveMQ, WebSphereMQ, and more. JMS clients connect to the JMS server differently for different JMS providers, and require different connection strings. For example:

- SonicMQ connection requires the SonicMQ Broker URL such as `tcp://myhost:2506`
- ActiveMQ connection requires the ActiveMQ broker URL such as `tcp://myhost:61616`
- WebSphereMQ connection requires the QueueManager name as it is configured in the JNDI namespace.

As the Progress OpenEdge JMS Adapter behaves as a proxy client for the JMS connection, the ABL API that creates connection with the JMS provider must provide an appropriate input for the provider.

The following figure shows the architectural relationships between the Progress OpenEdge JMS Adapter and the JMS provider Broker in BrokerConnect mode:



If you wish to wish to migrate your applications from SonicMQ to another JMS provider such as ActiveMQ or WebSphereMQ, you must modify your ABL-JMS API to point to the appropriate MQ Broker URL as follows:

Update WebSphereMQ connection strings

For WebSphereMQ:

```
RUN setBrokerURL IN psSessionH (INPUT "SampleQMgr").
```

where `SampleQMgr` is the name of the QueueManager or the connection factory for the queue or the topic that you want to use.

Note:

You can create the Connection Factory in the WebSphere MQClient Mode as `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory`. When an ABL client tries to use the Connection Factory that you created, make sure that the ABL Session type and the corresponding `ConnectionFactory` that you are using to connect are compatible as listed below:

- Use `jmsSession` with `ConnectionFactory`, `TopicConnectionFactory` or `QueueConnectionFactory`.
- Use `pubsubSession` with only `TopicConnectionFactory`
- Use `ptpSession` with only `QueueConnectionFactory`

Update ActiveMQ connection strings

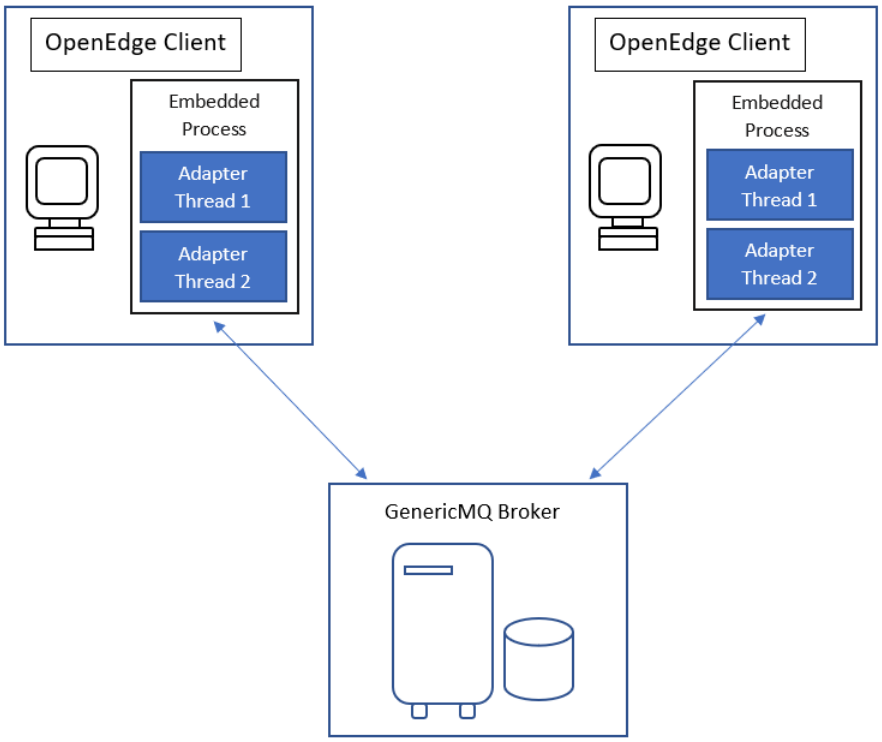
For ActiveMQ:

```
RUN setBrokerURL IN psSessionH (INPUT "myhost:61616").
```

where `61616` is the port at which the ActiveMQ Broker URL is listening.

Progress OpenEdge JMS Adapter ClientConnect

ClientConnect is a process that runs with your ABL client session. The following figure shows the architectural relationships between the Progress OpenEdge JMS Adapter and the JMS provider Broker in ClientConnect mode:



To configure the Progress OpenEdge JMS Adapter for ClientConnect mode, follow these steps:

1. Ensure your JMS provider is running on an accessible machine. For example, ActiveMQ should be running on port 8161 and be accessible at the following URL: `localhost:8161`.
2. Connect to the ptpsession using the -SMQConnect connection option. For more information, see Connection options on page 55.

The following code example shows how to send messages to an ActiveMQ server using ClientConnect:

```
DEFINE VARIABLE ptpsession AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.
DEFINE VARIABLE lDebug AS LOGICAL.
DEFINE VARIABLE cdate AS CHARACTER NO-UNDO FORMAT "x(16)".
DEFINE VARIABLE ddate AS DATE      NO-UNDO.


/* Creates a session object. */
RUN jms/ptpsession.p PERSISTENT SET ptpsession ("-SMQConnect").

/* Set user credentials. */
RUN setBrokerURL      IN ptpsession (INPUT "tcp://localhost:61616").
RUN setUser           IN ptpsession (INPUT "admin").
RUN setPassword       IN ptpsession (INPUT "admin").

/* Connect to the broker. */
RUN beginSession IN ptpsession.

/* Create a text message */
RUN createTextMessage IN ptpsession (OUTPUT messageH).
ddate = DATE(cdate).
RUN setText IN messageH ("This is test message sent.").
```

```
/* Publish the message on the "RequestQ" topic */
RUN sendToQueue IN ptpsession ("MyQueue1", messageH, ?, ?, ?).
RUN deleteMessage IN messageH.
RUN deleteSession IN ptpsession.

Message "Sent".
```

Manage connections and sessions

In order to use the messaging capabilities of JMS and the Progress OpenEdge JMS Adapter, a connection must be created to the JMS Provider Broker. An active connection receives messages. A session is a single-threaded context for sending and receiving messages. Since ABL is single-threaded, there is no compelling reason for multiple sessions per connection, nor for exposing the distinction between sessions and connections. In the context of the ABL–JMS API, the term *session* refers to the combination of a session and a connection.

Note: In JMS, a Java client can create several sessions per connection.

When more than one session per connection is required (for example, to send and receive messages concurrently), a second session is used implicitly in the Progress OpenEdge JMS Adapter, transparent to the ABL programmer.

The following topics describe:

- Create a JMS session on page 54
- Delete a JMS session on page 55
- Connection options on page 55
- Set and get JMS connection and session attributes on page 57
- Connect to the Progress OpenEdge JMS Adapter on page 58
- Establish session control on page 58
- Access message delivery parameters on page 58
- Request/Reply on page 59

Create a JMS session

These are the general steps to create a JMS session in ABL:

1. Run `jms/jmssession.p`, `jms/pubsubsession.p` or `jms/ptpsession.p` persistently with the Progress OpenEdge JMS Adapter connection parameters as `INPUT CHAR` parameters.
2. (Optional) Set JMS attributes and parameters by calling internal procedures in the session procedure.
3. Start the actual JMS connection and session by calling the beginSession procedure on page 172.

Note: Session attributes cannot be modified after calling the beginSession procedure on page 172.

An OpenEdge application can create and connect to multiple Session objects concurrently. You must create separate sessions to connect to each domain (PTP or Pub/Sub) with a separate Progress OpenEdge JMS Adapter Broker or a single Broker's unified domain.

Note: It is recommended that you use the JMS session domain and minimize the number of Session objects. Each session represents a separate JMS-provider client session and you want to minimize the number of JMS-provider client sessions.

Delete a JMS session

An application calls the deleteSession procedure on page 183 in the Session object to close and delete the session. This call terminates the underlying JMS connection and sessions, disconnects the OpenEdge client from the Progress OpenEdge JMS Adapter, deletes all the Message Consumer objects, and deletes the session's persistent procedure.

The deleteSession procedure on page 183 call does not delete the ABL Message objects associated with the session; those messages remain for possible use with other sessions.

Connection options

The Progress OpenEdge JMS Adapter supports most of the same connection options as the Progress Application Server for OpenEdge. You specify the desired options as the value of the *adapterConnection* parameter. The following table lists and explains the valid formats for expressing these options.

Table 1: Progress OpenEdge JMS Adapter connection options

Connection option	Description
`-H [`*`host_name`*` | `*`IP-address`*`]`	Specifies the network address that the oemessaging script runs on; or, if you use the -DirectConnect option, the address of a machine hosting the Progress OpenEdge JMS Adapter. You can use either the TCP/IP host name or the IP address of the machine. (Defaults to `localhost`.)
`-S [`*`port-number`*` | `*`service-name`*` ]`	Specifies the UDP port number that the oemessaging script runs on; or, if you use the -DirectConnect option, the TCP/IP port number for the Progress OpenEdge JMS Adapter. You can use either an explicit port number or a name associated with a port number in the TCP/IP services file. (Defaults to 3620 for a direct connection to a Progress OpenEdge JMS Adapter.)
`-DirectConnect`	If included, causes the -H and -S parameters to be interpreted as the network address and TCP/IP port number of a specific Progress OpenEdge JMS Adapter. If the -DirectConnect switch is omitted, the -H and -S parameters are interpreted as the network address and UDP port number of the Progress OpenEdge JMS Adapter.
`-ssl`	If included, specifies a secure connection to an SSL-enabled OpenEdge JMS Adapter. For more information, see *OpenEdge Getting Started: Core Business Services*.
`-nohostverify`	If included, turns off host verification for a secure (SSL) connection. In the case of an SSL connection, unless this switch is included, the client compares the host name (specified in the -H parameter or the -URL parameter) with the Common Name specified in the server certificate, and raises an error if they do not match. With -nohostverify in effect, the client never raises the error. This option works only in the context of a secure connection; that is, in combination with the -ssl switch or with HTTPS. For more information, see *OpenEdge Getting Started: Core Business Services*.
`-nosessionreuse`	If included, prevents the application from reusing the session ID when reconnecting to the same SSL-enabled Progress OpenEdge JMS Adapter .This option works only in the context of a secure (SSL) connection; that is, in combination with the -ssl switch or with HTTPS. For more information, see *OpenEdge Getting Started: Core Business Services*.
`-pf` *`pathname`*	

Connection option	Description
	Specifies a text file containing any of the application server connection parameters described in this table. Any other OpenEdge startup parameters in the file are ignored.
`-SMQConnect`	Creates a direct connection to the JMS-provider Broker using the `jmssession.p` Session object.

The following table shows several examples of valid *`adapterConnection`* parameters.

Table 2: Connection parameter examples

Connection parameter	Description
`""`	By default, connection to the Progress OpenEdge JMS Adapter running on UDP port 3620 on localhost.
`"-H host1 -S 5163"`	Connection to the Progress OpenEdge JMS Adapter running on UDP port 5163 on the machine host1.
`"-H fortress -S 3621 -DirectConnect -ssl -nohostverify"`	Secure connection directly to the SSL-enabled Progress OpenEdge JMS Adapter running on TCP/IP port 3621 on the machine fortress, with host verification disabled.
`RUN jms/jmssession.p PERSISTENT SET sessionH ("-SMQConnect").`	Connects directly to the JMS-provider Broker allowing better connection and session management

See *OpenEdge Getting Started: Core Business Services* for a comprehensive discussion of SSL connections, management of keys and certificates, and other security considerations in OpenEdge.

Set and get JMS connection and session attributes

After creating the session procedure, the application specifies connection and session attributes and retrieves values related to connection and session attributes. The following table lists the methods for handling connection and session attributes.

Table 3: Connection and session attributes

Setting	Getting
setAdapterService procedure on page 245	getAdapterService function on page 186
setJMSServerName procedure on page 264	getJMSServerName function on page 206
setBrokerURL procedure on page 247	getBrokerURL function on page 187

Setting	Getting
setPingInterval procedure on page 164	getConnectionMetaData function on page 191
setUser procedure on page 280	getUser function on page 226
setPassword procedure on page 270	For more information, see MultipartMessage on page 64 and the discussion of the getPassword on page 215 in the MultiPartMessage example.
setClientID procedure on page 250	getClientID function on page 190
setTransactedReceive procedure on page 279	getTransactedReceive function on page 225
setTransactedSend procedure on page 279	getTransactedSend function on page 225

Connect to the Progress OpenEdge JMS Adapter

After setting the previously described attributes as required, the application starts the JMS session and connection using the beginSession procedure on page 172.

Establish session control

The following table lists the methods an application uses to manage session control.

Table 4: Setting session methods

Method	Description
startReceiveMessages procedure on page 281	Starts receiving messages after creating a new session or after calling stopReceiveMessages procedure on page 282
stopReceiveMessages procedure on page 282	Causes the SonicMQ Broker to stop delivering messages to the OpenEdge client
deleteSession procedure on page 183	Closes a session and its underlying connection and deletes the session procedure
getConnectionID function on page 191	Returns the AppServer connection ID

Access message delivery parameters

Message delivery parameters set on the Session object are used as defaults for all messages sent in that session. The default can be changed by setting the parameters of the call to publish procedure on page 232, the call to sendToQueue procedure on page 244, or the call to requestReply procedure on page 241. These values cannot be changed after the beginSession procedure on page 172 is called.

The following table lists the methods for setting and getting delivery parameters.

Table 5: Setting and getting delivery parameters

Setting	Getting
setDefaultPriority procedure on page 256 setDefaultTimeToLive procedure on page 257 setDefaultPersistency procedure on page 256	getDefaultPriority function on page 197 getDefaultTimeToLive function on page 198 getDefaultPersistency function on page 197

Set the maximum number of messages

The default maximum number of active JMS messages in an OpenEdge session is 50. This is the maximum number of messages that have been created, but not deleted by the application. To change the default, the main procedure of the OpenEdge application must include the definition of JMS-MAXIMUM-MESSAGES global variable on page 230.

Note: If you exceed the limit on the JMS-MAXIMUM-MESSAGES global variable on page 230, an error is returned.

Discardable messages

The Progress OpenEdge JMS Adapter supports a `DISCARDABLE` message delivery mode. For non-transacted Pub/Sub sessions, `DISCARDABLE` delivers all messages to subscribers that are keeping up with the flow of messages, but drops the oldest messages waiting for lagging subscribers when new messages arrive, under any of the following conditions:

- When the message server's internal buffers for that subscriber session are full
- When a neighbor cluster member containing a Topic subscription is unavailable and a subscriber is located on the other cluster member
- When an intended durable subscriber is unavailable

An application controls message delivery mode using the publish procedure on page 232 and the setDefaultPersistency procedure on page 256.

Request/Reply

Request/Reply is a mechanism for the `JMSReplyTo` message header field to specify the destination where replies to a message should be sent. To specify the message destination, use the requestReply procedure on page 241.

Note: The term *destination* refers to both topics and queues.

Java–JMS supports a manual approach through the `JMSReplyTo` field, whereas the ABL–JMS implementation automates the request/reply sequence by:

- Sending a reply by setting the `reply` output parameter of the message handler
- Requesting a reply by calling the requestReply procedure on page 241 with a handle to a Message Consumer for the reply

The ABL–JMS implementation uses a temporary destination for the reply. It is an error to set the `JMSReplyTo` field of the message explicitly if requestReply procedure on page 241 is used. The reply is received by the Message Consumer asynchronously, just like any other message. The temporary destination is deleted when the Message Consumer object is deleted.

Work with messages

JMS-providers support several types of messages with different formats. Each message type represents the message body in a different format.

The following topics describe:

- Manage the lifecycle of an ABL - JMS message on page 60
- Create, populate, and access messages on page 61
- Send messages to a queue on page 69
- Publish messages to a topic on page 69
- Clear messages on page 70
- Delete messages on page 70
- Access message header properties on page 70
- Access message properties on page 71

Manage the lifecycle of an ABL - JMS message

An ABL–JMS message has a lifecycle for sending and a lifecycle for receiving.

Manage the sending lifecycle of an ABL - JMS message

To manage the sending lifecycle of an ABL–JMS message, follow these steps:

1. Create a message by running one of the following procedures in the Session object: createBytesMessage procedure on page 176, createDataSetMessage procedure on page 176, createHeaderMessage procedure on page 177, createMapMessage procedure on page 177, createMultipartMessage procedure on page 144, createStreamMessage procedure on page 179, createTempTableMessage procedure on page 180, or createXMLMessage procedure on page 181.
2. Populate a message by running `set...` and `write...` for header and data information.
3. Send the message to a destination.
4. Run the deleteMessage procedure on page 182 to delete the message.

Manage the receiving lifecycle of an ABL - JMS message

To manage the receiving lifecycle of an ABL–JMS message, follow these steps:

1. Receive a message in a Message Consumer object.
2. Run `get...` and `read...` to extract header information and body data.
3. Run the deleteMessage procedure on page 182 to delete the message.

Create, populate, and access messages

The Session object provides a method for creating each type of message. Each message provides methods for setting the content of the message body. The following topics describe:

- TextMessage on page 61
- HeaderMessage on page 62
- MapMessage on page 62
- StreamMessage on page 63
- BytesMessage on page 64
- MultipartMessage on page 64
- XMLMessage on page 65
- DataSetMessage on page 66
- TempTableMessage on page 66
- Java Object messages on page 66
- Message size limits on page 67
- Storing and extracting data on page 67

TextMessage

A `TextMessage` is a message type whose body contains text data.

The following table lists the methods for handling text messages.

Table 6: Methods for handling text messages

Create method	Populate method	Access method
createTextMessage procedure on page 181	setText procedure on page 278 setLongText procedure on page 267 appendText procedure on page 171 reset procedure on page 242 clearBody procedure on page 174	endOfStream function on page 185 getCharCount function on page 189 getText function on page 223 getTextSegment function on page 224 getLongText function on page 210 getLongTextCP function on page 210

For any `TextMessage` smaller than 32K, text data can be extracted and stored in a message by the getText function and the setText procedure method. For a `TextMessage` longer than 32K, the setLongText procedure and the getLongText function are available. Use of these calls is recommended for new code developed to process large character strings.

Note: You can continue to use the appendText procedure and the getTextSegment function provided in earlier versions of the OpenEdge Adapter for SonicMQ, when ABL imposed a 32K character limit on text strings. However, programming with these older calls is significantly more complex than using the newer setLongText procedure and the getLongText function.

The appendText procedure and the getTextSegment function concatenate text segments. With multiple appendText procedure calls, an OpenEdge client can create a `TextMessage` up to the limit of the JMS server. The JMS non-OpenEdge client receives a single `TextMessage` resulting from the concatenation of all the text segments.

The Progress OpenEdge JMS Adapter divides the received `TextMessage` into text segments of 8K (8192) or fewer characters. An application can then use multiple getTextSegment function calls to retrieve these segments. If getText function is called instead, the ABL–JMS API returns all of the text, and a run-time error occurs if the `TextMessage` is too large for the ABL interpreter to handle. An application can use the getCharCount function call to determine the total number of characters in a message.

For example, if the message value is `UNKNOWN`, or `""`, or a String of 5,000 characters, an application can use one getText function call (or one getTextSegment function call). If the message size is 16,400 characters, the first two getTextSegment function calls return 8192 characters each, and the last getTextSegment function call returns 16 characters.

The endOfStream function function returns true when all of the segments are retrieved (that is, when the number of getTextSegment function calls matches the number of segments). The setText procedure call implicitly calls clearBody procedure before setting the new text. The reset procedure and getText function calls transfer the message from write-only to read-only mode and position the message cursor before the first segment.

For more information, see Read-only and write-only modes on page 69.

Note: The 8K segment size is guaranteed. An OpenEdge application need not use the endOfStream function on page 185 for messages smaller than 8K, since there is only one segment. For information about code page conversions and text size limits, see XML code page encoding.

HeaderMessage

A `HeaderMessage` is a header-only message type that handles bodyless JMS messages. Use the createHeaderMessage procedure on page 177 to handle header messages. See Access message header properties on page 70 for information on methods that access message header information.

MapMessage

A `MapMessage` is a message type that contains a set of name/value pairs where values are Java primitives.

The following table lists the methods for handling map messages.

Table 7: Methods for handling map messages

Create method	Populate method	Access method
createMapMessage procedure on page 177	setBoolean procedure on page 246 setByte procedure on page 248 setBytesFromRaw procedure on page 249 setChar procedure on page 250 setDate procedure on page 251 setDateTime procedure on page 253 setDateTime-TZ procedure on page 254 setDouble procedure on page 257 setFloat procedure on page 259 setInt procedure on page 260 setInt64 procedure on page 261 setLong procedure on page 266 setLongString procedure on page 267 setShort procedure on page 275 setString procedure on page 277 clearBody procedure on page 174	getMapNames function on page 211 getItemType function on page 201 getBytesToRaw function on page 188 getChar function on page 188 getDate function on page 192 getDateTime function on page 193 getDateTime-TZ function on page 195 getDecimal function on page 196 getInt function on page 199 getInt64 function on page 200 getLogical function on page 208 getLongString function on page 209 getLongStringCP function on page 209

StreamMessage

A `StreamMessage` is a message type that allows applications to send and receive an unspecified number of items; each item is a Java data type. All basic Java data types are supported. When receiving any arbitrary Java data type, an application uses methods to read and specify an ABL data type. When writing a message from ABL, an application uses methods to send any of those Java data types and to specify the data.

The following table lists the methods for handling stream messages.

Table 8: Methods for handling stream messages

Create method	Populate method	Access method
createStreamMessage procedure on page 179	writeBoolean procedure on page 284 writeByte procedure on page 285 writeBytesFromRaw procedure on page 286 writeChar procedure on page 286 writeDate procedure on page 287 writeDateTime procedure on page 288 writeDateTime-TZ procedure on page 289 writeDouble procedure on page 289 writeFloat procedure on page 290 writeInt procedure on page 290 writeInt64 procedure on page 291 writeLong procedure on page 292 writeLongString procedure on page 292 writeShort procedure on page 293 writeString procedure on page 294 reset procedure on page 242 clearBody procedure on page 174	endOfStream function on page 185 moveToNext procedure on page 231 readBytesToRaw procedure on page 233 readChar function on page 234 readDate function on page 234 readDateTime function on page 235 readDateTime-TZ function on page 236 readDecimal function on page 236 readInt function on page 237 readInt64 function on page 238 readLogical function on page 238 readLongString function on page 239 readLongStringCP function on page 239

BytesMessage

A `BytesMessage` is a message type that contains an uninterpreted stream of bytes.This message type allows the passing of data "as is" without any interpretation by the ABL–JMS API or the JMS server.

The following table lists the methods for handling bytes messages.

Table 9: Methods for handling bytes messages

Create method	Populate method	Access method
createBytesMessage procedure on page 176	setMemptr procedure on page 268 writeBytesFromRaw procedure on page 286 reset procedure on page 242 clearBody procedure on page 174	readBytesToRaw procedure on page 233 getMemptr function on page 212 endOfStream function on page 185 getBytesCount function on page 187

To write data to a `BytesMessage`, an application uses `RAW` or `MEMPTR` variables with writeBytesFromRaw procedure or setMemptr procedure. To read data, it uses readBytesToRaw procedure or getMemptr function.

Note: The `RAW` data type has a 32K size limit. To bypass this limit, an application uses the writeBytesFromRaw procedure and the readBytesToRaw procedure repeatedly. The `MEMPTR` data type does not have a 32K limit. To access `MEMPTR` bytes data, an application uses the setMemptr procedure and the getMemptr function.

For an example, see Publishing, subscribing, and receiving an XML document in a BytesMessage .

For example, a `BytesMessage` can pass an XML document encoded in a code page that does not match the OpenEdge client's code page.

For more information, see XML code page encoding . For an example, see Publishing, receiving, and parsing an XMLMessage .

MultipartMessage

A `MultipartMessage` is a message type that contains one or more discreet parts. A part can be a SonicMQ message, Character data, or Byte data. Parts are identified by a unique content ID character value and can be accessed by ID or index. Each part also contains a content-type value for identifying the data in the part. For message parts, the content-type is defined by Sonic and represents each message type supported by Sonic. A bytes part or text part has a user-defined content-type. There are no restrictions on what this content-type can be, but it is recommended that you use standard MIME types such as text/XML for XML data and text/plain for character data.

The following table lists the methods for handling multi-part messages.

Table 10: Methods for handling multi-part messages

Create method	Populate method	Access method
createMultipartMessage procedure on page 144	addMessagePart procedure on page 142 addBytesPart procedure on page 170 addTextPart procedure on page 170 appendText procedure on page 171 clearBody procedure on page 174	isMessagePart function on page 229 getBytesPartByIndex function on page 145 getBytesPartByID function on page 145 getContentType on page 148 getPartCount function on page 215 getTextPartByID function on page 223 getTextPartByIndex function on page 224 getMessagePartByIndex function on page 213 getMessagePartByID function on page 212

Note: The RAW data type has a 32K size limit. To bypass this limit, an application uses the writeBytesFromRaw procedure and the readBytesToRaw procedure repeatedly. The MEMPTR data type does not have a 32K limit. To access MEMPTR bytes data, an application uses the setMemptr procedure and the getMemptr function.

XMLMessage

An XMLMessage is a message type whose body contains a well-formed XML document (a SAX-WRITER, SAX-READER, or X-DOCUMENT). ABL has built in support to send and receive XML messages. The following table lists the methods for handling XML messages.

Table 11: Methods for handling XML messages

Create method	Populate method	Access method
createXMLMessage procedure on page 181	setText procedure on page 278 setLongText procedure on page 267 appendText procedure on page 171 reset procedure on page 242 setX-Document procedure on page 280 setSaxReader procedure on page 274 deleteSaxWriter procedure on page 183 clearBody procedure on page 174	endOfStream function on page 185 getCharCount function on page 189 getText function on page 223 getTextSegment function on page 224 getLongText function on page 210 getLongTextCP function on page 210 getSaxWriter function on page 221 getX-Document function on page 226

The XMLMessage is an extension of a JMS TextMessage. XMLMessage supports the same methods as TextMessage.

XML messages can be used in conjunction with the ABL XML parser:

- **Incoming messages** — Parse the XML text using the getX-Document function or the setSaxReader procedure
- **Outgoing messages** — Save the XML text using the setX-Document procedure or the getSaxWriter function

It is important to consider the *code page* of XML messages. (A code page is a table that maps each character on it to a unique numeric value.) Theoretically, any code page can be used to encode XML documents. However, each XML parser supports some or all code pages, and XML parsers differ with respect to the code page conversions that they can do.

With the ABL–JMS API, the conversion rules are straightforward. The text stored in an XML message by the OpenEdge application is expected to be encoded in the internal code page of the OpenEdge client (the `-cpinternal` startup parameter). For more information on the `-cpinternal` startup parameter, see *OpenEdge Deployment: Startup Command and Parameter Reference*.

The ABL–JMS implementation automatically converts the text to *Unicode* when a SonicMQ XML message is created. Unicode is an encoding format that provides a unique number for every character, regardless of platform, program, or language. The ABL–JMS implementation also converts the Unicode text received in XML messages to the internal code page of the OpenEdge client when the text is extracted.

For more information, see XML code page encoding .

DataSetMessage

A `DataSetMessage` is a message type whose body contains a ProDataSet. The information sent and received is based upon the existing `XMLMessage`. ABL has built in support to transform a ProDataSet into XML. The following table lists the methods for handling DataSet messages.

Table 12: Methods for handling DataSet messages

Create method	Populate method	Access method
createDataSetMessage procedure on page 176	setDataSet procedure on page 251 reset procedure on page 242 clearBody procedure on page 174	getDataSet function on page 192

For an example of a `DataSetMessage`, see DataSetMessage on page 99 .

TempTableMessage

A `TempTableMessage` is a message type whose body contains a temp-table. The information sent and received is based upon the existing `XMLMessage`. ABL has built in support to transform a temp-table into XML. The following table lists the methods for handling TempTable messages.

Table 13: Methods for handling TempTable messages

Create method	Populate method	Access method
createTempTableMessage procedure on page 180	setTempTable procedure on page 278 reset procedure on page 242 clearBody procedure on page 174	getTempTable function on page 222

For an example of a `TempTableMessage`, see TempTableMessage on page 96 .

Java Object messages

The ABL–JMS API does **not** support Java Object messages. If a Java Object message is received on behalf of an OpenEdge client, the client's asynchronous error handler receives a `TextMessage` with the header of the Java Object message and a text body with the string "ObjectMessage: Not Supported." (For more information, see Error and condition handling on page 79 .)

Message size limits

There is no limit to the ABL message size. However, ABL imposes a 32K limit on each item of a `StreamMessage` or `MapMessage`. For more information about text size limits, see XML code page encoding .

JMS-providers often do not have a hard-coded maximum message size; the largest tested message is 1MB.

When using very large messages (exceeding 1MB), you might need to modify the JVM's memory limit values, specified in the `jvmArgs` property of the `AdminServerPlugins.properties` file. For example, if the OpenEdge Adapter for SonicMQ fails with an `OutofMemory` error in the log, you should modify the arguments for the sizes of the memory heap (`-mx`) and the stack (`-ss`). The following sample entry specifies 40MB for the memory heap and 8MB for the stack:

```
jvmArgs= –Xmx40m –Xss8m
```

Note: You can use the `mergeprop` utility installed with OpenEdge to manually edit the `AdminServerPlugins.properties` file. For information on using `mergeprop`, see *OpenEdge Getting Started: Installation and Configuration*.

Storing and extracting data

When writing data to a message, an application uses the name of the data type to specify the Java data type in the message; the ABL name is identical to the Java name. For example, Java uses the writeShort procedure on page 293 to write a number to a `StreamMessage` as *`short`*. The ABL counterpart is the internal procedure `writeShort(INTEGER)`.

In the context of extracting data from a message, there is an important difference between the ABL model and the Java model with respect to the names of the methods:

- In Java, the name of the method determines the data type to which the extracted data is converted. For example, readLongString function on page 239 extracts a value (for example, an `INTEGER` value from a `StreamMessage`) and converts it to a `String` value.
- In ABL, the equivalent function is the readChar function on page 234 to convert a value (such as an `INTEGER` value) to an ABL `CHARACTER` value.

The following table maps the ABL data types to the JMS data types for data storage.

Table 14: JMS and ABL data types for storing data

ABL data type	JMS data type
LOGICAL	boolean
INTEGER	byte
INTEGER	short
INTEGER	int
INT64	long
DECIMAL	long

ABL data type	JMS data type
DECIMAL	float
DECIMAL	double
CHARACTER	String
LONGCHAR	String
A single CHARACTER	char
RAW	byte array
MEMPTR	byte array (only with BytesMessage)
DATE	String
DATETIME	String
DATETIME-TZ	String

The following table maps the available conversions from JMS data types to ABL data types for data extraction.

Table 15: JMS and ABL data types for extracting data

JMS data type	ABL data type
boolean	LOGICAL or CHARACTER
byte	INTEGER, DECIMAL, or CHARACTER
short	INTEGER, DECIMAL, or CHARACTER
int	INTEGER, DECIMAL, or CHARACTER
long	INT64, DECIMAL or CHARACTER
float	DECIMAL or CHARACTER
double	DECIMAL or CHARACTER
String	CHARACTER or LONGCHAR
char	CHARACTER
byte array	RAW or MEMPTR (MEMPTR is available only with BytesMessage)
Java date string	DATE, DATETIME, or DATE-TZ

Read-only and write-only modes

As in Java–JMS, the `StreamMessage`, `TextMessage`, `XMLMessage`, and `BytesMessage` are created in write-only mode. In write-only mode, an application can use only data-setting methods, not data-extraction methods.

The reset procedure on page 242 puts the cursor before the first item of the message and transfers it to read-only mode.

Note: The publish procedure on page 232 , sendToQueue procedure on page 244 , and requestReply procedure on page 241 call the reset procedure implicitly.

The message is received by the receiver in read-only mode. The clearBody procedure on page 174 clears the message body and transfers the message to write-only mode.

Note: Read-only and write-only refer to the body of the message, not its header. Read-only and write-only modes do not apply to Header messages, which lack a body.

Unlike in Java–JMS, a `MapMessage` in the ABL–JMS implementation is always in read/write mode; there is no read-only or write-only mode for a `MapMessage`.

Note: The reset procedure has no effect when called on Map and Header messages.

---

clearBody and clearProperties

The clearBody procedure and clearProperties procedure, supported by all message types, function as follows:

- The clearBody procedure on page 174 deletes all data from the message body.
- The clearProperties procedure on page 174 deletes all header properties (but not the JMS-predefined header fields).

Send messages to a queue

In the PTP domain, applications send messages to a queue. To send a message to a queue with Java–JMS, an application obtains a handle to a queue object, creates a Queue Sender object, and uses the queue sender to send messages. Sending a message to a queue with the ABL–JMS API involves these general steps:

1. The application calls the sendToQueue procedure on page 244 in the `ptpsession.p` or `jmssession.p` object.
2. The application specifies the queue name as an `INPUT` parameter of type `CHARACTER`.

The application can set other sending parameters (such as `persistency`, `timeToLive`, and `priority`) in the Session object as a default for all the messages it sends, or it can set these parameters at each sendToQueue procedure call.

Publish messages to a topic

In the Pub/Sub domain, applications publish messages to topics. To publish a message with Java–JMS, an application obtains a handle to a Topic object and creates a Publisher object. It then uses the Publisher Object to publish messages. Publishing a message to a topic with the ABL–JMS API involves these general steps:

1. The application publishes messages through the publish procedure on page 232 of the `pubsubsession.p` or `jmssession.p` object.
2. The application specifies the topic name as an `INPUT` parameter of type `CHARACTER`.

The application can set other sending parameters (such as `persistency`, `timeToLive`, and `priority`) in the Session object as a default for all the messages it sends, or it can set these parameters at each publish procedure call.

Clear messages

An application clears the body of a message, leaving header and property values unchanged, using the clearBody procedure on page 174 . The clearProperties procedure on page 174 deletes all header properties (but not the JMS-predefined header fields).

Delete messages

An application explicitly deletes a message using the deleteMessage procedure on page 182 . For example:

```
RUN deleteMessage IN messageH.
```

Access message header properties

The message header provides envelope information about a message. The message header interface is supported by all message types, and all message types have the same header information.

The message header is not created directly by the application. When any type of message is created, its header procedure is automatically created. The message procedure delegates header method calls to its header procedure.

The following table lists the methods to access message header information.

Table 16: Accessing message header information

Set header methods	Get header methods
setJMSReplyTo procedure on page 263 setReplyToDestinationType procedure on page 273 setJMSCorrelationID procedure on page 262 setJMSCorrelationIDAsBytes procedure on page 263 setJMSType procedure on page 265	getJMSReplyTo function on page 205 getReplyToDestinationType function on page 219 getJMSCorrelationID function on page 201 getJMSCorrelationIDAsBytes function on page 202 getJMSType function on page 207 getJMSDestination function on page 203 getJMSRedelivered function on page 205 getMessageType function on page 214 getJMSMessageID function on page 204 getJMSDeliveryMode function on page 202 getJMSTimestamp function on page 206 getJMSExpiration function on page 203 getJMSPriority function on page 204 hasReplyTo function on page 227

Access message properties

Message properties can add more envelope information about a message. The number of header fields is fixed, but properties are flexible. An application can add any number of property name-and-value pairs. The following table lists the methods to access message properties.

Table 17: Accessing message properties

Setting message properties	Getting message properties
setBooleanProperty procedure on page 247 setByteProperty procedure on page 248 setDateProperty procedure on page 252 setDateTimeProperty procedure on page 254 setDateTimeTzProperty procedure on page 255 setDoubleProperty procedure on page 258 setFloatProperty procedure on page 260 setIntProperty procedure on page 261 setInt64Property procedure on page 262 setLongProperty procedure on page 266 setShortProperty procedure on page 275 setStringProperty procedure on page 277 clearProperties procedure on page 174	getCharProperty function on page 189 getDateProperty function on page 193 getDateTimeProperty function on page 194 getDateTimeTzProperty function on page 195 getDecimalProperty function on page 196 getIntProperty function on page 199 getInt64Property function on page 200 getLogicalProperty function on page 208 getPropertyType function on page 217 ggetPropertyNames function on page 216

The requesting application clears the properties of a message (keeps header and body values unchanged) using clearProperties procedure.

Consume messages

A Message Consumer consumes messages either asynchronously or synchronously. The message consumption type is set for a session. Messages are processed by the Message Consumer when the ABL Virtual Machine (AVM) is in a `WAIT-FOR` state or other IO-blocking state. While the application is in such a state, all other UI and non-UI events are handled normally. `WAIT-FOR` can be called explicitly by the ABL code. It can also be called through the waitForMessages procedure on page 283 on the Session object, which works the same for GUI, character, batch, and PAS for OpenEdge applications. Once a message is consumed, the content of the message is inaccessible.

The following sections describe:

- Create a Message Consumer object on page 72
- Create a message handler process on page 72
- Set reply properties on page 73
- Receive messages from a queue on page 73
- Subscribe to a topic on page 73
- Terminate the Message Consumer object on page 73
- Process messages on page 74
- Control flow of messages on page 74

- Reuse messages on page 74
- Message-reception issues on page 74
- Reply mechanisms on page 75

Create a Message Consumer object

The OpenEdge application uses a Message Consumer object to receive messages from a destination or to receive asynchronous error messages. In a Session object, the application creates a Message Consumer object using the createMessageConsumer procedure on page 178 .

The lifecycle of a Message Consumer object includes these general steps:

1. An application implements a procedure to handle the messages.
2. The application creates the Message Consumer, specifying the message-handling procedure.
3. The application uses the Message Consumer object to do one of the following: subscribe to a topic (Pub/Sub) or receive messages from the queue (PTP); set an error handler and receive error messages asynchronously from the JMS-provider through the Progress OpenEdge JMS Adapter; or receive replies in a request/reply cycle.
4. After using the Message Consumer object, the application can activate it by getting into a `WAIT FOR` state (or any IO-blocking state where the application processes events).
5. When the Message Consumer finishes processing all messages of interest, the application calls the deleteConsumer procedure on page 182 to release the resources in the OpenEdge application, Progress OpenEdge JMS Adapter, and the JMS-provider Broker.

Create a message handler process

A message handler processes the incoming message from the Message Consumer. When an incoming JMS or error message is received, the message handler is called automatically so that the application can process the message. The ABL programmer creates a message handler using the messageHandler procedure on page 230 . The OpenEdge application passes context to the message handler using the setApplicationContext procedure on page 245 .

Access message handler information

The following table lists procedures for getting message handler properties and type of message being handled.

Table 18: Methods for the message handler

Getting message handler properties	Type of message handled
getApplicationContext function on page 186 getDestinationName function on page 198 getProcHandle function on page 216 getProcName function on page 216 getReplyAutoDelete function on page 217 getReplyPersistency function on page 218 getReplyPriority function on page 218 getReplyTimeToLive function on page 219 getSession function on page 221	inErrorHandling function on page 227 inMessageHandling function on page 228 inQueueBrowsing function on page 228 inReplyHandling function on page 229

Set reply properties

The Message Consumer sets the reply properties by using the setReplyPriority procedure, setReplyTimeToLive procedure, setReplyPersistency procedure, and setReplyAutoDelete procedure.

Receive messages from a queue

In the PTP domain, applications receive messages from a queue. The application calls the receiveFromQueue procedure on page 240 in `ptpsession.p` or `jmssession.p` with the queue name and a handle to the Message Consumer object. The application can pass a JMS properties selector expression to the receiveFromQueue procedure call to specify which messages the receiver wants to receive from the queue.

Queue browsing

The PTP model supports *queue browsing*, a mechanism that lets an application view the content of messages in a queue without actually consuming (receiving) the messages. The ABL–JMS API supports queue browsing through the browseQueue procedure on page 172 in the PTP Session object or JMS Session object.

The messages can be handled by the message handler in the same way as messages coming from a receiveFromQueue procedure on page 240 call, but they are not acknowledged and are not subject to the transactional context of the session.

Subscribe to a topic

In the Pub/Sub domain, applications subscribe to topics of interest. The application calls the subscribe procedure on page 282 in `pubsubsession.p` or `jmssession.p` with the topic name and a handle to the Message Consumer object. The application implements a message-handling routine for handling the incoming messages, as well as a Message Consumer object that contains the message handler and provides context to the application when it processes messages.

Durable subscriptions

A subscriber typically receives messages while it is active. Some applications might require that a subscriber receives all messages even if the subscriber is inactive when the messages are published. In order to meet this requirement, you can create a *durable subscriber*. A *durable subscription* guarantees message delivery. A durable subscription is registered with the JMS-provider Broker with a unique identity; the broker retains the subscription's messages until they are received by the application or until they expire. The application can pass a JMS properties selector expression to the subscribe procedure on page 282 to specify which messages the subscriber wants to receive. The application can also specify whether it wants to receive its own published messages. Use the cancelDurableSubscription procedure on page 173 to cancel a durable subscription.

Terminate the Message Consumer object

In a Session object, the application deletes the Message Consumer object and releases resources in the OpenEdge application, the Progress OpenEdge JMS Adapter, and the JMS-provider Broker using the deleteConsumer procedure on page 182.

Process messages

To control message processing use the waitForMessages procedure on page 283.

Control flow of messages

The application controls the flow of messages to the JMS-provider client from a queue using the setPrefetchCount procedure on page 165 and the setPrefetchThreshold procedure on page 166.

Note: When the Progress OpenEdge JMS Adapter sends a message to a queue that is full or to a topic that is full, an error is raised.

Reuse messages

The application sets message reuse using the setReuseMessage procedure on page 274 and getReuseMessage function on page 220.

Message-reception issues

The topics that follow discuss several message-reception issues in the PTP and Pub/Sub domains.

Stop and start message reception

To actually start receiving messages, the OpenEdge application must call the startReceiveMessages procedure on page 281 in the Session object. One call to the startReceiveMessages procedure is sufficient for the session. The application typically calls the startReceiveMessages procedure after subscribing to all topics of interest (in the Pub/Sub domain) or registering Message Consumer objects with the queues of interest (in the PTP domain).

The application can also call the stopReceiveMessages procedure on page 282 to temporarily stop the reception of messages. To resume message reception, it can call the startReceiveMessages procedure again.

In the Pub/Sub domain, calling the stopReceiveMessages procedure does not cancel existing subscriptions; however, for any nondurable subscription, messages published while reception is stopped are not delivered.

In the PTP domain, the messages are queued while the client is in the stopReceiveMessages procedure state and are delivered to the client after the startReceiveMessages procedure is called again.

Stopping the reception of messages is recommended when an application is not going to process messages for a while.

Note: After calling the stopReceiveMessages procedure, the OpenEdge client might receive one message sent from the server prior to execution of the call.

Caution: `StopReceiveMessages` procedure should not be invoked in a message handler.

Message Consumer scope

A Message Consumer object can be used to handle only one subscription (in the Pub/Sub domain) or receive messages from only one queue (in the PTP domain).

When the deleteConsumer procedure on page 182 is called, message reception is canceled and the Message Consumer object is deleted.

Note: To delete a durable subscription (in the Pub/Sub domain), the cancelDurableSubscription procedure on page 173 in `pubsubsession.p` or `jmssession.p` must be called as well, since deleteConsumer procedure only suspends the subscription in the current session. There is no equivalent to a durable subscription in the PTP domain. It is an error to call the cancelDurableSubscription procedure while there is an active Message Consumer for that subscription. First call the deleteConsumer procedure to delete the Message Consumer.

When a Message Consumer object is used for receiving replies through the requestReply procedure on page 241 call, it can be used many times; there is no need to create one for every call. The deleteSession procedure on page 183 call deletes all Message Consumer objects for that session.

OpenEdge run-time message-processing states

An OpenEdge application receives and processes messages when it is in an I/O-blocking state. The same rules that determine when asynchronous completion procedures are fired also determine when message handlers are called. The OpenEdge application should typically use the `WAIT-FOR` statement or the waitForMessages procedure API session call for processing messages as well as for other events.

The waitForMessages procedure on page 283 is a convenient way to write message-handling code that is independent of the environment in which the OpenEdge application is executed (GUI, CHUI, batch, or PAS for OpenEdge). It processes all events that occur while the application is waiting, including user-interface events and asynchronous call events, and it allows the application to specify when to stop waiting.

The waitForMessages procedure takes three input parameters: a procedure handle, the name of a user-defined function in the procedure that returns a `LOGICAL` value, and a `timeOut` parameter of type `INTEGER` (specifying an interval in seconds). The waitForMessages procedure waits and processes events as long as: a) the user-defined function returns `TRUE`; and b) the interval specified by the `timeOut` value elapses without any messages being received.

The user-defined function is evaluated by the ABL–JMS API after the message handler is executed. Typically, the OpenEdge application should have logic for changing the return value of the function in the message handler.

Synchronous message reception

ABL does not explicitly support receiving messages synchronously, but the same effect can be achieved by use of the `WAIT-FOR` statement or the waitForMessages procedure on page 283. These constructs wait for a user-defined event. When the desired message is received, the message handler can trigger the termination of the `WAIT-FOR` statement or the waitForMessages procedure —for example, by applying the specified user-defined event.

Reply mechanisms

This section applies to both the Pub/Sub and the PTP domains.

Java–JMS provides no built-in mechanism for replies. It is the responsibility of the application to:

- Designate a Destination object (typically a temporary destination) for replies
- Send this Destination object to the receiver (typically through the **ReplyTo** field in the *message header*, a set of fields containing values to identify and route the message)

The receiver must extract the reply destination from the message and follow the normal publish (or send) steps to reply.

The ABL–JMS API simplifies this process, both for the sender needing a reply and for the receiver needing to reply:

- **Sender** — The ABL–JMS API requestReply procedure on page 241 can publish messages in the same way as the publish procedure, or send messages to a queue in the same way as the sendToQueue procedure. In addition, a Message Consumer object for replies is passed to the requestReply procedure as an input parameter. The ABL–JMS implementation automatically routes all the replies to that Message Consumer object. See Request/Reply on page 59 for additional information.
- **Receiver** — To reply, the message receiver returns a *reply* message handle as an output parameter in the message-handling routine. The application can call the setReplyPersistency procedure on page 271 in the Message Consumer object to automatically delete replies after sending them.

An application can also publish a reply message or send it to a queue by first calling the getReplyToDestinationType function on page 219 to extract the name of the reply destination, and then calling the publish procedure or sendToQueue procedure directly.

Note: If the **ReplyTo** destination is a temporary destination, an application must send a reply before deleting the original message. Deleting the original message tells the ABL–JMS implementation that the `ReplyTo` temporary destination will no longer be used.

By default, the type of the **ReplyTo** destination matches the type of the origin of the message:

- If the message was created by a Pub/Sub Session object, the value of the **ReplyTo** field is considered a topic name
- If the message was created by a PTP Session object, the value of the **ReplyTo** field is considered a queue name

However, it is legal to designate a queue for replying to a published message, or a topic for replying to messages received from a queue. To accommodate this, the ABL–JMS API supports the setReplyToDestinationType procedure on page 273 and the getReplyToDestinationType function on page 219, both of which support the `CHARACTER` values `topic` and `queue`.

The setReplyToDestinationType procedure can be called if the OpenEdge application calls the setJMSReplyTo procedure and sets a destination from a domain other than that of the session. The getReplyToDestinationType function must be called when the OpenEdge application receives a message and wants to reply to it, but is not certain about the **ReplyTo** domain.

Transacted sessions

A *transacted session* allows an application to send or receive groups of messages as one atomic operation:

- A session that is transacted for sending guarantees that either all messages in a group are sent, or none is sent
- A session that is transacted for receiving guarantees that a group of received messages are acknowledged only after all messages in the group are successfully processed

The following table lists the methods available for controlling the execution and recovery of transactions.

Table 19: Managing transaction attributes

Setting	Rolling back
commitSend procedure on page 175 commitReceive procedure on page 175	rollbackSend procedure on page 243 rollbackReceive procedure on page 243

The typical Java–JMS transacted application uses two sessions, one for transacted sending and one for transacted receiving. The ABL–JMS implementation uses two JMS sessions behind the scenes, but at the ABL API level, there is only one Session object.

The application controls whether sending, receiving, or both are transacted. It makes the session transacted by calling the setTransactedSend procedure on page 279, the setTransactedReceive procedure on page 279, or both in the Session object.

A session that is transacted for sending, receiving, or both is constantly in a transaction mode. When a transaction is committed or rolled back, a new one is automatically started.

Transacted sending

When an application calls the commitSend procedure on page 175 in a Session object, all messages that have been published or sent to a queue with the current transaction are sent. When an application calls the rollbackSend procedure on page 243e in a Session object, all such messages are discarded.

Transacted receiving

When an application calls the commitReceive procedure on page 175 in a Session object, all messages that have been received with the current transaction are acknowledged. When an application calls the rollbackReceive procedure on page 243 in a Session object, all such messages are re-received (yielding the same effect as calling the recover procedure on page 241 in a nontransacted session).

Illegal calls: recover and setNoAcknowledge

Since message acknowledgement and recovery are handled automatically in a transacted session, it is an error to call the recover procedure and setNoAcknowledge procedure in a session that is transacted for receiving.

ABL transactions and JMS transacted sessions

ABL transactions and JMS transactions are not integrated. For example, a `DO TRANSACTION` block might be rolled back while the JMS calls inside the transaction block are committed. The OpenEdge application must synchronize between ABL transactions and JMS transactions.

Note: For information about the handling of errors and error conditions, see Error and condition handling on page 79 .

Message acknowledgement, forwarding, and recovery

A client sends an *acknowledgement* to tell the JMS-provider Broker that the client received and processed a message and does not need to receive that message again. Acknowledgement of a message prevents the message and all previous messages from being delivered to that session again.

The following table lists the methods the application uses to set message acknowledgement.

Table 20: Acknowledging and forwarding messages

Setting	Getting
setSingleMessageAcknowledgement procedure on page 169 setNoAcknowledge procedure on page 269 acknowledgeAndForward procedure on page 141	getSingleMessageAcknowledgement function on page 155 getNoAcknowledge function on page 214

The sections that follow describe automatic message acknowledgement, preventing message acknowledgement, and message recovery.

Automatic message acknowledgement

With the ABL–JMS API, an incoming message is acknowledged automatically when the message handler finishes execution. Acknowledgement is sent on the request for the next message, improving performance.

A client or communication failure between the time the message handler finishes execution and the time the ABL–JMS implementation sends the acknowledgement can result in the messages being redelivered (according to the JMS message redelivery rules). An application can use a transacted session to avoid this message redelivery problem.

Unlike Java–JMS, the ABL–JMS API does not support the explicit acknowledgement of messages or the "lazy" acknowledgement of messages (the JMS `CLIENT_ACKNOWLEDGE` and `DUPS_OK_ACKNOWLEDGE` modes).

Preventing message acknowledgement

An OpenEdge application can explicitly prevent acknowledgement of a message by calling the setNoAcknowledge procedure of the Message Consumer object. (The Message Consumer object is passed as a parameter to the message-handling procedure.) The setNoAcknowledge procedure is typically used when the application wants to receive the same message again because of an error in processing it, or when receipt of a group of messages is to be signaled by explicit acknowledgement of only the last message in the group.

Single-message acknowledgement

Normally, an OpenEdge client application automatically acknowledges a message when the message handler procedure completes. In `SINGLE_MESSAGE_ACKNOWLEDGE` mode, however, each message requires its own acknowledgement; if you choose not to acknowledge a message, it is never acknowledged.

To turn on `SINGLE_MESSAGE_ACKNOWLEDGE` mode, an OpenEdge client application calls the setSingleMessageAcknowledgement procedure of the session handle with the input parameter set to `TRUE`. To turn off this mode, the application calls the same method with the input parameter set to `FALSE`.

Message recovery

If an application wants to receive all unacknowledged messages again, it can call the recover procedure on page 241 in the Session object. If the recover procedure is called on a session stopped by the stopReceiveMessages procedure on page 282, the session is recovered and message delivery is restarted.

Example

Consider the following scenario:

1. A client retrieves a message from a broker's queue.
2. The broker wants to be notified when the message reaches its ultimate destination.
3. The ultimate destination is a remote queue.
4. The client sends the message on its way.

To acknowledge receipt of a message whose ultimate destination is a remote queue, you might enclose the message and acknowledgement in a single transaction; but this introduces the overhead and complexity of transaction processing. SonicMQ provides a cleaner solution, embodied in the following steps:

1. Run the setSingleMessageAcknowledgement procedure to set the session to `SINGLE_MESSAGE_ACKNOWLEDGE`.
2. Run the acknowledgeAndForward procedure within the message event handler, specifying a destination queue name, the original message handle, and optional message-delivery properties (priority, time to live, and persistency). If the method is successful, the message is acknowledged and forwarded in a single atomic operation.

Error and condition handling

This topic provides information about handling of errors and conditions with the ABL–JMS API. To configure logging, see Configure logging for the Progress OpenEdge JMS adapter on page 48.

From the point of view of the ABL programmer, there are two types of errors and conditions, programming errors and run-time conditions:

- A *programming error* is an erroneous sequence of calls to the ABL–JMS API, or the calling of the API with invalid parameters. Typically, programming errors should not occur in a deployed application. An example of a programming error is an attempt by the application to make a TextMessage call such as setText procedure, setLongText procedure, and appendText procedure in a StreamMessage. Programming errors should be tracked down and fixed at development time. The primary source of information for that phase is the ABL–JMS API (see ABL - JMS API Reference).
- A *run-time condition* is an event that disturbs the normal flow of the application. Such events can occur in a deployed application, so the ABL programmer should try to handle them programmatically. Examples of run-time conditions include attempts to connect to a JMS server that is not currently running, and attempts to subscribe to a topic without the proper authorization. For information about programmatically handling run-time conditions, refer to the API reference of your JMS provider.

A second criterion for classifying errors and conditions is whether the problem is reported by the ABL–JMS implementation synchronously or asynchronously:

- A problem is reported **synchronously** if it occurs and is detected while the OpenEdge application is executing an ABL–JMS API call.
- A problem is reported **asynchronously** when it comes from the asynchronous error reporting system of the JMS server (OnException Events) or from the ABL–JMS mechanism that delivers messages asynchronously to the OpenEdge client.

Programming errors are usually reported synchronously. Run-time conditions are reported either synchronously or asynchronously.

Note: For a complete description of ABL error handling, see *OpenEdge Development: ABL Error Handling*.

Handling errors

To manage errors use the setErrorHandler procedure on page 258 and setNoErrorDisplay procedure on page 269.

Synchronously reported errors and conditions

Errors are reported synchronously when something goes wrong at a method call. The problems can be either programming errors or run-time conditions. Examples include attempts to publish to an unauthorized topic or attempts to receive from a nonexistent queue.

An ABL API function reports problems synchronously by returning an `Unknown value (?)`.

Some programming errors are not detected by the ABL–JMS API but rather by the ABL interpreter. For example, an attempt to call the setText procedure in a StreamMessage causes error `6456`:

```
Procedure message.p has no entry point for setText. (6456)
```

To report a problem synchronously, the ABL–JMS API internal procedure calls:

```
RETURN ERROR <error-message>
```

This call raises an error condition at the caller. The caller can use regular ABL techniques to handle the error: a `NO-ERROR` phrase or an `ON ERROR` block, coupled with checking the `RETURN-VALUE` value to obtain the error message. If an application uses the `NO-ERROR` phrase, it must check the `STATUS-ERROR:error` flag to determine whether a problem has occurred.

By default, every synchronously reported error or condition is displayed by the ABL–JMS API, which calls:

```
MESSAGE <error-message> VIEW-AS ALERT-BOX.
```

This mechanism allows a quick analysis and resolution of the problem at development time. At deployment time, however, the application developer might want to handle problems programmatically and prevent the message from appearing. Calling the setNoErrorDisplay procedure on page 269 in the Session object suppresses the message display.

Note: Message objects inherit the `display/noDisplay` property from the session that created them. However, after a message is created, it is independent of the session. The setNoErrorDisplay procedure on page 269 must be called in the Message object itself to change this property.

Asynchronously reported conditions

Typically, problems reported asynchronously are run-time conditions, such as the failure of the SonicMQ Broker or the failure of communication between the Progress OpenEdge JMS Adapter and the JMS-provider Broker. Another example is the failure to send an automatic reply (the message handler is set with a reply message, but the JMS-provider fails to send the reply).

The error condition is reported in a `TextMessage`, with several possible `CHAR` message properties in the message header: `exception`, `errorCode`, `linkedException-1`, `linkedException-2`… `linkedException-`*n* (where *n* is a number of additional exceptions linked to the main exception). Use the getPropertyNames function on page 216 to get a list of properties in the error message header. See Messaging Examples for an example.

The application should handle problems of this type programmatically by creating a Message Consumer object and passing it to the setErrorHandler procedure in the Session object. If an application does not set an error handler, a default error handler displays the error message and the properties in alert boxes.

Note: An application must call the beginSession procedure before creating the error-handling Message Consumer object and calling the setErrorHandler procedure on page 258.

Run-time conditions

Typically, run-time exceptions are generated by the Java–JMS code on the server. In such cases, the format of the error message obtained from the `RETURN-VALUE` is:

```
<java-exception>:<error-message>.
```

The ABL programmer can look up the types of exceptions thrown by the JMS-provider and handle some of them programmatically. The most typical run-time error conditions are connection and authorization failures.

Connection and communication failures

The most common run-time error condition is a connection failure. The beginSession procedure on page 172, which creates the connection to the Progress OpenEdge JMS Adapter and the JMS server, reports connection failures synchronously by calling:

```
RETURN ERROR <error-message>.
```

The error can result from a failure to connect either to the Progress OpenEdge JMS Adapter or to the JMS server. If the connection to the JMS server fails, the format of the error message is:

```
<java-exception>:<error-message>.
```

A communication failure that occurs after a successful connection might be detected:

- Synchronously (for example, when the application is trying to publish a message)
- Asynchronously through the error handler

It might take several minutes for the timeout mechanism to trigger a communication failure event.

If you are using SonicMQ as a JMS provider, you can detect potential communication failures more quickly using the setPingInterval procedure on page 164 (a SonicMQ extension) to instruct the Progress OpenEdge JMS Adapter to actively ping the SonicMQ Broker every *n* seconds.

Message handler errors and conditions

A message-handling procedure is an arbitrary ABL program, and the programmer is free to use any ABL technique to handle problems that occur during the processing of a message. However, the following issues and limitations exist:

- Message handlers should handle ERROR, STOP, and QUIT conditions and not propagate them. An unhandled condition is considered a programming error.
- Since the message handler returns control to the ABL–JMS implementation and the message handler cannot raise a condition, there must be a mechanism to allow the message handler to communicate problems to the rest of the OpenEdge application. You can use the setApplicationContext procedure on page 245 call to pass an ABL procedure handle to the Message Consumer object. The message handler can obtain the procedure handle by calling the getApplicationContext function on page 186 in the Message Consumer object and can then make the appropriate internal procedure calls.
- As mentioned in Message acknowledgement, forwarding, and recovery on page 77, the message handler can call the setNoAcknowledge procedure on page 269 of the Message Consumer to prevent the message from being acknowledged in a session that is not transacted for receiving.
- Calling WAIT-FOR is allowed inside a message handler, but no further messages from that Session object are received until the message handler returns.
- The following recursive calls from the message handler into the ABL–JMS API of the same Session object are considered programming errors: deleteSession procedure on page 183, deleteConsumer procedure on page 182, and recover procedure on page 241. There are no restrictions on calling these API entries of another Session object.

Interrupts

An interrupt (**CTRL+C** on UNIX or Microsoft platforms) while an ABL–JMS call is executing can cause the call to return either an ABL STOP condition or an ERROR condition, depending on the exact timing. The ABL–JMS implementation guarantees that partial messages will not be sent or received as the result of an interrupt.

Progress OpenEdge JMS Adapter failure

If communication with the Progress OpenEdge JMS Adapter is lost, or if the Progress OpenEdge JMS Adapter shuts down while the OpenEdge client is performing a WAIT-FOR or waitForMessages procedure on page 283 statement, an ABL STOP condition is raised.

If communication with the Progress OpenEdge JMS Adapter is lost, or if the Progress OpenEdge JMS Adapter shuts down while the ABL–JMS implementation is actively trying to communicate to it (for example, when the OpenEdge application calls the publish procedure or the subscribe procedure), an ERROR or STOP condition is raised, depending on the exact point at which the failure is discovered.

PTP message example

A PTP messaging example consists of the basic steps described this topic.

Creating a PTP session

In the following session example, the application creates a session object by calling `ptpsession.p` persistently.

```
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
RUN jms/ptpsession.p PERSISTENT SET hPTPSession ("-H localhost -S 5162" ).
```

Creating the session object specifies the connection parameters to the JMS-provider Broker. This allows an application to set different session-level attributes before starting the JMS session. The connection to the JMS-provider Broker and the JMS session does not occur until the application calls the beginSession procedure on page 172.

Connecting to the broker

In the following connection example, the OpenEdge application connects to the JMS-provider Broker to begin exchanging messages.

```
RUN setBrokerURL IN hPTPSession ("tcp//machinename:2506").
RUN beginSession IN hPTPSession.
```

Creating a Message Consumer

The OpenEdge client requires a queue for sending messages. You create a queue using the JMS-provider console (for example, Sonic Management Console). Queues must be defined before starting your session. Then you create a Message Consumer to receive requests from queue, as follows:

```
RUN createMessageConsumer IN hPTPSession
  (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).
```

Preparing to receive messages

The OpenEdge application begins listening on the queue and prepares to receive messages from the queue, as follows:

```
RUN receiveFromQueue IN hPTPSession ("myQueue", ?, hConsumer).
RUN startReceiveMessages IN hPTPSession.
/* Wait to receive the messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
```

The Message Consumer (`hConsumer` handle) listens on `myqueue` and handles messages using the `myintproc` internal procedure. The startReceiveMessages procedure on page 281 starts receipt of incoming messages.

Sending messages to the queue

The application sends a message to the queue using the sendToQueue procedure on page 244, as follows:

```
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
/* Code to create message */
RUN sendToQueue IN hPTPSession ("myQueue", hMessage, ?, ?, ?)
```

Note: The queue must be created on the JMS-provider management console.

Receiving messages from the queue

The Message Consumer receives a message from the queue and executes the business logic.

Deleting a message

The application deletes the messages after it finishes using them, as shown in the following example.

```
RUN deleteMessage IN hMessage.
```

PTP Message Summary

To see the full sample code for sending and receiving a message using a PTP session, see PTP message example summary on page 84.

PTP message example summary

Complete code for sending a message using a PTP session

The following sample summarizes the steps for sending a message.

```
/* Sending a message to myqueue */
DEFINE VARIABLE hMessage    AS HANDLE NO-UNDO.
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
/* Creates PTP session*/
RUN  jms/ptpsession.p PERSISTENT SET hPTPSession ("-H localhost -S 5162 -DirectConnect"
 ).
/*Connects to the broker */
RUN setBrokerURL IN hPTPSession ("tcp//machinename:2506").
RUN beginSession IN hPTPSession.
/* Create a message */
RUN create...Message IN hPTPSession (OUTPUT hMessage).
RUN set... IN hMessage ("Message").
/*Send the message to "myqueue" */
RUN sendToQueue IN hPTPSession ("myQueue", hMessage, ?, ?, ?)
/* Delete message and session */
RUN deleteMessage IN hMessage.
RUN deleteSession IN hPTPSession.
```

Complete code for receiving a message using a PTP session

The following sample summarizes the steps for receiving a message.

```
/* Receives a message from myqueue. */
DEFINE VARIABLE hConsumer   AS HANDLE NO-UNDO.
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
/* Creates PTP session*/
RUN jms/ptpsession.p PERSISTENT SET hPTPSession ("-H localhost -S 5162 -DirectConnect"
 ).
/*Connects to the broker */
RUN setBrokerURL IN hPTPSession ("tcp//machinename:2506").
RUN beginSession IN hPTPSession.
/* Messages received from myqueue are handled by the "myintproc" procedure. */
RUN createMessageConsumer IN hPTPSession
  (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).
RUN receiveFromQueue IN hPTPSession ("myQueue", ?, hConsumer).
RUN startReceiveMessages IN hPTPSession.
/* Wait to receive the messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
/* Delete session */
RUN deleteSession IN hPTPSession.
PROCEDURE myintproc:
  DEFINE INPUT  PARAMETER hMessage  AS HANDLE NO-UNDO.
  DEFINE INPUT  PARAMETER hConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hReply    AS HANDLE NO-UNDO.
  /* Business logic here */
  . . .
  /* Delete message */
  RUN deleteMessage IN hMessage.
  APPLY "U1" TO THIS-PROCEDURE.
END.
```

Pub/Sub messaging example

A Pub/Sub messaging example consists of the basic steps described in this topic.

Creating a Pub/Sub session procedure

In the following example, the application creates a session object by calling `pubsubsession.p` persistently.

```
DEFINE VARIABLE hPTPSession AS HANDLE NO-UNDO.
RUN jms/pubsubsession.p PERSISTENT
  SET hPubSubSession ("-H localhost -S 5162 ").
```

Creating the session object specifies the connection parameters to the JMS-provider Broker. This allows an application to set different session-level attributes before starting the JMS session. The connection to the JMS-provider Broker and the JMS session does not occur until the application calls the beginSession procedure on page 172.

Connecting to the broker

The OpenEdge application connects to the JMS-provider Broker to begin exchanging messages, as follows:

```
RUN setBrokerURL IN hPubSubSession ("tcp://machinename:2506").
RUN beginSession IN hPubSubSession.
```

Creating a Message Subscriber

You create a message subscriber to receive the message from the topic `newtopic`. The subscriber handles the message using the internal procedure `myintproc`, as shown in the following example.

```
/* Receives requests from the newTopic */
RUN createMessageConsumer IN hPubSubSession
  (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).
```

Note: Topics can be configured at run time.

Subscribing to a topic

Applications subscribe to topics of interest. To subscribe to a topic, the application subscribes to a topic and prepares to receive messages from the topic, as shown in the following example.

```
/* Subscribes to newtopic */
RUN SUBSCRIBE IN hPubSubSession ("newTopic", ?, ?, NO, hConsumer).
/* Start receiving requests */
RUN startReceiveMessages IN hPubSubSession.
/* Wait to receive the messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
```

Publishing to a topic

An application uses the publish procedure to publish messages to a topic, as shown in the following example.

```
DEFINE VARIABLE hMessage AS HANDLE NO-UNDO.
/* Code to create message */
RUN publish IN hPubSubSession ("newTopic", hMessage, ?, ?, ?).
```

Consuming messages from a topic

The Message Consumer receives a message from the topic and executes the business logic.

Deleting a message

The application deletes the messages after it finishes using them, as shown in the following example.

```
RUN deleteMessage IN hMessage.
```

Pub/Sub messaging example summary

To see the full sample code for sending and receiving a message using a publisher/subscriber model, see Pub/sub messaging example summary on page 87.

Pub/sub messaging example summary

Complete code for publish a message using a Pub/Sub session

The following sample summarizes the steps for sending a message.

```
/* Publishes a message to newtopic. */
DEFINE VARIABLE hMessage       AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates the Pub/Sub session */
RUN  jms/pubsubsession.p PERSISTENT
  SET hPubSubSession ("-H localhost -S 5162 ").
/*Connects to the broker */
RUN setBrokerURL IN hPubSubSession ("tcp://machinename:2506").
RUN beginSession IN hPubSubSession.
/* Create a message */
RUN create...Message IN hPTPSession (OUTPUT hMessage).
RUN set... IN hMessage ("Message").
/* Publish the message on the "newTopic" topic */
RUN publish IN hPubSubSession ("newTopic", hMessage, ?, ?, ?).
/* Delete message and session */
RUN deleteMessage IN hMessage.
RUN deleteSession IN hPubSubSession.
```

Complete code for receiving a message using a Pub/Sub session

The following sample summarizes the steps for receiving a message.

```
/* Subscribes and receives a message from myTopic. */
DEFINE VARIABLE hConsumer      AS HANDLE NO-UNDO.
DEFINE VARIABLE hPubSubSession AS HANDLE NO-UNDO.
/* Creates the Pub/Sub session. */
RUN jms/pubsubsession.p PERSISTENT
  SET hPubSubSession ("-H localhost -S 5162 ").
/*Connects to the broker */
RUN setBrokerURL IN hPubSubSession ("ltcp://machinename:2506").
RUN beginSession IN hPubSubSession.
/* Subscriptes to the newTopic topic. Received messages are handled by the
   myintproc internal procedure. */
RUN createMessageConsumer IN hPubSubSession
  (THIS-PROCEDURE, "myintproc", OUTPUT hConsumer).
/* Subscribes to newtopic */
RUN SUBSCRIBE IN hPubSubSession ("newTopic", ?, ?, NO, hConsumer).
/* Start receiving requests */
RUN startReceiveMessages IN hPubSubSession.
/* Wait to receive the messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
/* Delete session */
RUN deleteSession IN hPTPSession.
PROCEDURE myintproc:
  DEFINE INPUT  PARAMETER hMessage  AS HANDLE NO-UNDO.
  DEFINE INPUT  PARAMETER hConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hReply    AS HANDLE NO-UNDO.
  /* Business logic here */
  . . .
  /* Delete message. */
  RUN deleteMessage IN hMessage.
  APPLY "U1" TO THIS-PROCEDURE.
END.
```

Programming scenarios

The following topics describe additional programming considerations and scenarios:

- Using JMS 1.1 unified domain model on page 88
- Implement ClientConnect on page 89
- Enhanced XML support on page 89
- TempTableMessage on page 96
- DataSetMessage on page 99

Using JMS 1.1 unified domain model

Prior to OpenEdge Release 10.1, OpenEdge clients were required to create a JMS session that was either for PTP or Pub/Sub messaging. If the client needed to use both queues and topics, it was necessary to create two JMS sessions.

Currently, OpenEdge supports JMS 1.1. JMS 1.1 unifies the two messaging domains into one domain. Therefore, OpenEdge clients may utilize both PTP or Pub/Sub messaging within the same JMS session. You can access both queues and topics using the same JMS session object. The following example shows the ABL code for using queues and topics in the same JMS session object:

```
RUN jms/jmssession.p PERSISTENT SET hSession (adapterConnection).
```

Note: The ModChat example demonstrates using the `jmssession` to perform both PTP and Pub/Sub messaging. The example uses a serialized connection object and server-based message selectors. For information on locating the examples, see the OpenEdge messages. For an alphabetical API reference, see ABL - JMS API Reference.

Implement ClientConnect

OpenEdge clients can connect directly to a JMS-provider Broker by using the Progress OpenEdge JMS Adapter for a messaging session. By connecting directly to the JMS-provider Broker, the OpenEdge client has better control over connection management, and there is no need to manage and configure a Progress OpenEdge JMS Adapter server process.

Additional benefits include the availability of client persistence and fault tolerance. For more information on client persistence, see Configure fault tolerance and high availability for ActiveMQ Artemis on page 104, Configure client persistence for SonicMQ on page 119, Configure fault tolerance for SonicMQ on page 121, or your JMS-provider's documentation.

Caution: This method creates a larger run-time footprint for your OpenEdge client.

Using SMQConnect on a client

In the following example, the application creates a session procedure by calling `jmssession.p` persistently specifying the `-SMQConnect` connection parameter.

```
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("MQBrokerHost:2506").
RUN beginSession IN hSession.
```

Note: Each messaging session creates a connection to the JMS-provider Broker.

Enhanced XML support

OpenEdge clients can send well-formed XML documents and additional types of data, such as temp-tables and ProDatasets, as `XMLMessage`. The `TempTableMessage` and `DataSetMessage` transport data to the JMS-provider Broker using XML. ABL has built-in functionality to transform `TEMP-TABLE` or `ProDataSet` data into XML. Additionally, OpenEdge clients read, write, and parse XML using `SAX-READER`, `SAX-WRITER`, and `X-DOCUMENT`.

For more information on accessing the examples files, see OpenEdge messages.

Saxwriter message

The following example shows how to use the SAX-WRITER object.

```
DEFINE VARIABLE hSaxWriter AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession   AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg      AS HANDLE NO-UNDO.
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN SetBrokerURL IN hSession ("-H localhost -S 5162 ").
RUN beginSession IN hSession.
RUN createXMLMessage IN hSession (OUTPUT hMesg).
/* Will write an envelope address */
hSaxWriter = DYNAMIC-FUNCTION('getSaxWriter':u IN hMesg, ?).
/* Want to format this so it is easy to read */
hSaxWriter:FORMATTED = TRUE.
hSaxWriter:START-DOCUMENT().
/* The ENCODING           attribute defaults to UTF-8 */
/* The FRAGMENT           attribute defaults to FALSE */
/* The STRICT             attribute defaults to TRUE  */
hSaxWriter:START-ELEMENT("mailingaddress").
RUN xmlData(INPUT "name", INPUT "Joe Perry").
hSaxWriter:START-ELEMENT("address").
/* This node contains an attribute */
hSaxWriter:INSERT-ATTRIBUTE("type", "personal").
RUN xmlData(INPUT "street",  INPUT "11 Sugerland St.").
RUN xmlData(INPUT "city",    INPUT "Somerville").
RUN xmlData(INPUT "state",   INPUT "MA").
RUN xmlData(INPUT "zipcode", INPUT "02143").
hSaxWriter:END-ELEMENT("address").
hSaxWriter:START-ELEMENT("address").
hSaxWriter:INSERT-ATTRIBUTE("type", "business").
RUN xmlData(INPUT "name",   INPUT "Progress Software").
RUN xmlData(INPUT "street", INPUT "14 Oak Park").
RUN xmlData(INPUT "city",   INPUT "Bedford").
RUN xmlData(INPUT "state",  INPUT "MA").
RUN xmlData(INPUT "zip",    INPUT "01730").
hSaxWriter:END-ELEMENT("address").
hSaxWriter:WRITE-EMPTY-ELEMENT("default").
hSaxWriter:INSERT-ATTRIBUTE("type", "personal").
hSaxWriter:END-ELEMENT("mailingaddress").
hSaxWriter:END-DOCUMENT().
RUN deleteSaxWriter IN hMesg (saxWriterH).
RUN sendToQueue IN hSession ("SampleQ1", hMesg, ?, ?, ?).
RUN deleteMessage IN hMesg.
PROCEDURE xmlData:
  DEFINE INPUT PARAMETER xmlNode  AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER charData AS CHARACTER NO-UNDO.
  hSaxWriter:START-ELEMENT(xmlNode).
  hSaxWriter:WRITE-CHARACTERS(charData).
  hSaxWriter:END-ELEMENT(xmlNode).
END PROCEDURE.
```

setSaxReader example

The following code sample shows how to use the setSaxReader procedure :

```
DEFINE VARIABLE hSax AS HANDLE NO-UNDO.
CREATE SAX-READER hSax.
/* SAX-READER setup as needed */
PROCEDURE messageHandler:
  DEFINE INPUT  PARAMETER hMessage         AS HANDLE NO-UNDO.
  DEFINE INPUT  PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hAutoReply       AS HANDLE NO-UNDO.
  DEFINE VARIABLE hResult AS HANDLE    NO-UNDO.
  DEFINE VARIABLE mType   AS CHARACTER NO-UNDO.
  mtype = DYNAMIC-FUNCTION('getMessageType':u IN hMessage).
  CASE mType:
    WHEN "TempTableMessage" THEN DO:
      hResult = DYNAMIC-FUNCTION('GetTempTable':u IN hMessage, ?, ?, ? ).
      /* TempTable actions as needed */
    END.
    WHEN "DatasetMessage" THEN DO:
      hResult = DYNAMIC-FUNCTION('GetDataSet':u IN hMessage, ?, ?, ? ).
      /* DataSet actions as needed */
    END.
    WHEN "XMLMessage" THEN DO:
      RUN setSaxReader IN hMessage (hSax).
      hSax:SAX-PARSE().
    END.
  END CASE.
  RUN deleteMessage IN hMessage.
END.
```

saxSender.p

The SAX-WRITER object reads XML from a file using the SAX-READER object and send it to a queue using an XMLMessage. The following example shows how to use the SAX-WRITER object.

```
DEFINE VARIABLE hdl1     AS HANDLE NO-UNDO.
DEFINE VARIABLE hdl2     AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg    AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
/* Start up the session to the JMS-provider broker */
RUN jms/ptpsession.p PERSISTENT SET hSession ("-H localhost -S 5162 ").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
/* Create the Sonic XML message */
RUN createXMLMessage IN hSession (OUTPUT hMesg).
/* The Adapter function GetSaxWriter will return a handle to a newly created
   SAX-WRITER and set its output to a destination local to the Adapter. The
   application may then make normal SAX-WRITER calls using this handle.*/
hdl2 = DYNAMIC-FUNCTION('GetSaxWriter':u IN hMesg, ?).
hdl2:START-DOCUMENT().
/* Create a SAX-READER to read in the xml file to be sent in a SonicMQ XML
   message.  The SAX-READER callback procedures will use the handle to the
   SAX-WRITER just created to copy the XML. */
CREATE SAX-READER hdl1.
hdl1:SET-INPUT-SOURCE("FILE", "personal.xml").
hdl1:SAX-PARSE().
DELETE OBJECT hdl1.
hdl2:END-DOCUMENT().
/* The Adapter procedure DeleteSaxWriter will copy the XML written with the
   SAX-WRITER calls into the SonicMQ XML message.  It will then delete the
   SAX-WRITER specified by the handle. */
RUN DeleteSaxWriter IN hMesg (hdl2).
/* Send the XML message. */
RUN sendToQueue IN hSession ("SampleQ1", hMesg, ?, ?, ?).
/* Disconnect the session from the SonicMQ broker */
RUN deleteMessage IN hMesg.
RUN deleteSession IN hSession.
/*************************************************/
/* callbacks for the SAX-READER function SAX-PARSE() */
/*************************************************/
/* Procedure is called when the parser finds a start tag for an element. */
PROCEDURE StartElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER cLocalName   AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER qname        AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER attributes   AS HANDLE    NO-UNDO.
  hdl2:START-ELEMENT(cLocalName, namespaceURI).
END PROCEDURE.
/* This callback gets passed the character data for an element. */
PROCEDURE Characters:
  DEFINE INPUT PARAMETER mCharData AS MEMPTR    NO-UNDO.
  DEFINE INPUT PARAMETER iNumChars AS INTEGER   NO-UNDO.
  DEFINE VARIABLE cData AS CHARACTER NO-UNDO.
  cData = GET-STRING(mCharData, 1, GET-SIZE(mCharData)).
  hdl2:WRITE-CHARACTERS(cData).
END PROCEDURE.
/* This callback is called when the parser finds the end tag for an Element. */
PROCEDURE EndElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER cLocalName   AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER qName        AS CHARACTER NO-UNDO.
  hdl2:END-ELEMENT(cLocalName, namespaceURI).
END.
```

saxReceiver.p

The SAX-READER object reads an XMLMessage from a queue and writes it to a LONGCHAR. The following example shows how to use the SAX-READER object.

```
DEFINE VARIABLE hdl2         AS HANDLE   NO-UNDO.
DEFINE VARIABLE hSession     AS HANDLE   NO-UNDO.
DEFINE VARIABLE lch          AS LONGCHAR NO-UNDO
  VIEW-AS EDITOR SIZE 70 BY 30 LARGE.
DEFINE VARIABLE hMsgConsumer AS HANDLE   NO-UNDO.
DEFINE VARIABLE hMesg        AS HANDLE   NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL  NO-UNDO INITIAL TRUE.
/* Start up the session to the JMS-provider broker */
RUN jms/ptpsession.p PERSISTENT SET hSession ("-H localhost -S 5162 ").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
/* Create the message consumer and start receiving messages. */
RUN createMessageConsumer IN hSession
  (THIS-PROCEDURE, "messagehandler", OUTPUT hMsgConsumer).
RUN receiveFromQueue IN hSession ("SampleQ1", ?, hMsgConsumer).
RUN startReceiveMessages IN hSession.
/* Wait for all messages to be received. */
RUN waitForMessages IN hSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hSession.
/* Message handler procedure */
PROCEDURE messageHandler:
  DEFINE INPUT  PARAMETER hMessage         AS HANDLE NO-UNDO.
  DEFINE INPUT  PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hAutoReply       AS HANDLE NO-UNDO.
  DEFINE VARIABLE hdl1 AS HANDLE NO-UNDO.
  CREATE SAX-WRITER hdl2.
  hdl2:SET-OUTPUT-DESTINATION("LONGCHAR",lch).
  hdl2:START-DOCUMENT().
  CREATE SAX-READER hdl1.
  /* The Adapter procedure SetSaxReader will set the input source for a
     SAX-READER to the XML message that has been received. The application may
     then use normal SAX-READER calls to access the XML from the message. */
  RUN SetSaxReader IN hMessage (hdl1).
  hdl1:SAX-PARSE().
  DELETE OBJECT hdl1.
  RUN deleteMessage IN hMessage.
  hdl2:END-DOCUMENT().
  DISPLAY lch.
  ASSIGN
    lch          = ""
    stillWaiting = FALSE.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
  RETURN stillWaiting.
END.
/*****************************************************/
/* callbacks for the SAX-READER function SAX-PARSE() */
/*****************************************************/
/* Procedure is called when the parser finds the start tag for an element. */
PROCEDURE StartElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER cLocalName   AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER qname        AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER attributes   AS HANDLE    NO-UNDO.
  hdl2:START-ELEMENT(cLocalName, namespaceURI).
END.
/* This callback gets passed the character data for an element.*/
PROCEDURE Characters:
  DEFINE INPUT PARAMETER charData  AS MEMPTR  NO-UNDO.
  DEFINE INPUT PARAMETER iNumChars AS INTEGER NO-UNDO.
  DEFINE VARIABLE data AS CHARACTER NO-UNDO.
  data = GET-STRING(charData, 1, GET-SIZE(charData)).
  hdl2:WRITE-CHARACTERS(data).
```

```
END PROCEDURE.
/* This callback is called when the parser finds the end tag for an Element. */
PROCEDURE EndElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER cLocalName   AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER qName        AS CHARACTER NO-UNDO.
  hdl2:END-ELEMENT(cLocalName, namespaceURI).
END.
```

setX-Document example

The following code sample shows how to use the setX-Document procedure on page 280.

```
CREATE X-DOCUMENT hdl1.
hdl1:LOAD("file", "4k.xml", false).
RUN createXMLMessage IN hSession (OUTPUT hMesg).
RUN setX-Document IN hMesg(hdl1).
RUN sendToQueue IN hSession ("SampleQ1", hMesg, ?, ?, ?).
RUN deleteMessage IN hMesg.
```

getX-Document example

The following sample code shows how to use getX-Document function on page 226.

```
PROCEDURE messageHandler:
  DEFINE INPUT  PARAMETER hMessage         AS HANDLE NO-UNDO.
  DEFINE INPUT  PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hAutoReply       AS HANDLE NO-UNDO.
  DEFINE VARIABLE mtype   AS CHARACTER NO-UNDO.
  DEFINE VARIABLE hResult AS HANDLE    NO-UNDO.
  mtype = DYNAMIC-FUNCTION('getMessageType':u IN hMessage).
  CASE mtype:
    WHEN "TemptableMessage" THEN DO:
      hResult = DYNAMIC-FUNCTION('GetTempTable':u IN hMessage, ?, ?, ? ).
      /* TempTable actions as needed */
    END.
    WHEN "DatasetMessage" THEN DO:
      hResult = DYNAMIC-FUNCTION('GetDataSet':u IN hMessage, ?, ?, ? ).
      /* DataSet actions as needed */
    END.
    WHEN "XMLMessage" THEN DO:
      hResult = DYNAMIC-FUNCTION('getX-Document':u IN hMessage).
      /* X-DOCUMENTcalls as needed */
    END.
  END CASE.
  RUN deleteMessage IN hMessage.
END.
```

domSender.p

The X-DOCUMENT object reads XML from a file and sends it to a queue using an XMLMessage. The following example shows how to use X-DOCUMENT object to send a message.

```
DEFINE VARIABLE hdl1     AS HANDLE NO-UNDO.
DEFINE VARIABLE hMesg    AS HANDLE NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
/* Start up the session to the JMS-provider broker */
RUN jms/ptpsession.p PERSISTENT SET hSession ("-H localhost -S 5162 ").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
/* Create the Sonic XML message */
RUN createXMLMessage IN hSession (OUTPUT hMesg).
/* Load the XML file. */
CREATE X-DOCUMENT hdl1.
hdl1:LOAD("file", "personal.xml", FALSE).
/* The Adapter function SetX-Document will copy the XML from the X-DOCUMENT
   into the XML message. */
RUN setX-Document IN hMesg(hdl1).
DELETE OBJECT hdl1.
/* Send the XML message. */
RUN sendToQueue IN hSession ("SampleQ1", hMesg, ?, ?, ?).
/* Disconnect the session from the SonicMQ broker */
RUN deleteMessage IN hMesg.
RUN deleteSession IN hSession.
```

domReceiver.p

The X-DOCUMENT object reads an XMLMessage from a queue and writes it to a LONGCHAR. The following example shows how to use X-DOCUMENT object to receive a message.

```
DEFINE VARIABLE hSession     AS HANDLE   NO-UNDO.
DEFINE VARIABLE lch          AS LONGCHAR NO-UNDO
  VIEW-AS EDITOR SIZE 70 BY 30 LARGE.
DEFINE VARIABLE hMsgConsumer AS HANDLE   NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL  NO-UNDO INITIAL TRUE.
/* Start up the session to the JMS-provider broker */
RUN jms/ptpsession.p PERSISTENT SET hSession ("-H localhost -S 5162 ").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
/* Create the message consumer and start receiving messages. */
RUN createMessageConsumer IN hSession
  (THIS-PROCEDURE, "messageHandler", OUTPUT hMsgConsumer).
RUN receiveFromQueue IN hSession ("SampleQ1", ?, hMsgConsumer).
RUN startReceiveMessages IN hSession.
/* Wait for all messages to be received. */
RUN waitForMessages IN hSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hSession.
/* Message handler procedure */
PROCEDURE messageHandler:
  DEFINE INPUT PARAMETER hMessage         AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hAutoReply      AS HANDLE NO-UNDO.
  DEFINE VARIABLE hdl1 AS HANDLE NO-UNDO.
  /* The Adapter function GetX-Document will return a handle to a newly
     created X-DOCUMENT and load it with the XML message that has been
     received. The application may then use normal X-DOCUMENT calls to access
     the XML from the message. */
  hdl1 = DYNAMIC-FUNCTION('GetX-Document':u IN hMessage).
  hdl1:SAVE("LONGCHAR",lch).
  DELETE OBJECT hdl1.
  RUN deleteMessage IN hMessage.
  DISPLAY lch.
  ASSIGN
    lch          = ""
    stillWaiting = FALSE.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
  RETURN stillWaiting.
END.
```

TempTableMessage

OpenEdge applications use temp-tables for data. The new TempTableMessage supports using temp-tables for JMS messaging. The XMLMessage is the basis for the TempTableMessage. A non-OpenEdge application receives a TempTableMessage as an XMLMessage. The JMS header property signals an OpenEdge application that the incoming message is a TempTableMessage.

For more information on accessing the examples files, see OpenEdge messages.

For an alphabetical API reference, see ABL - JMS API Reference.

ttmgs_send.p

The following example shows sending a `TempTableMessage`.

```
DEFINE VARIABLE hMessage AS HANDLE  NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE  NO-UNDO.
DEFINE VARIABLE numRecs  AS INTEGER NO-UNDO.
DEFINE VARIABLE ptp      AS LOGICAL NO-UNDO.
DEFINE VARIABLE ttH      AS HANDLE  NO-UNDO.
DEFINE TEMP-TABLE ttCustomer NO-UNDO LIKE customer.
/* Creates a session object. */
RUN  jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
FOR EACH customer NO-LOCK:
  CREATE ttCustomer.
  ASSIGN
    ttCustomer.CustNum    = customer.CustNum
    ttCustomer.Name       = customer.Name
    ttCustomer.Address    = customer.Address
    ttCustomer.Address2   = customer.Address2
    ttCustomer.City       = customer.City
    ttCustomer.State      = customer.State
    ttCustomer.PostalCode = customer.PostalCode
    ttCustomer.Country    = customer.Country
    numRecs               = numRecs + 1.
END.
ttH = TEMP-TABLE ttCustomer:HANDLE.
RUN createTemptableMessage IN hSession (OUTPUT hMessage).
RUN setTempTable IN hMessage (ttH, ?, TRUE).
IF ptp THEN
  RUN sendToQueue IN hSession ("SampleQ1", hMessage, ?, ?, ?).
ELSE
  RUN PUBLISH IN hSession ("TestTopic", hMessage, ?, ?, ?).
RUN deleteMessage IN hMessage.
DELETE ttCustomer.
MESSAGE "Number of records processed: " + STRING(numRecs).
RUN createTextMessage in hSession (OUTPUT hMessage).
RUN setText IN hMessage (STRING(numRecs)).
IF ptp THEN
  RUN sendToQueue IN hSession ("SampleQ1", hMessage, ?, ?, ?).
ELSE
  RUN PUBLISH IN hSession ("TestTopic", hMessage, ?, ?, ?).
RUN deleteMessage IN hMessage.
RUN deleteSession IN hSession.
```

ttmsg_recv.p

The following example shows receiving a `TempTableMessage`.

```
DEFINE VARIABLE hMsgConsumer AS HANDLE  NO-UNDO.
DEFINE VARIABLE hMesg        AS HANDLE  NO-UNDO.
DEFINE VARIABLE hSession     AS HANDLE  NO-UNDO.
DEFINE VARIABLE numRecsRead  AS INTEGER NO-UNDO.
DEFINE VARIABLE ptp          AS LOGICAL NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL NO-UNDO INITIAL TRUE.
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
RUN createMessageConsumer IN hSession
  (THIS-PROCEDURE,   /* this procedure will handle it */
   "messageHandler", /* name of internal procedure */
   OUTPUT hMsgConsumer).
IF ptp THEN
  RUN receiveFromQueue IN hSession ("SampleQ1", ?, hMsgConsumer).
ELSE
  RUN subscribe IN hSession
    ("TestTopic",
     ?,    /* durable subscription */
     ?,    /* no message selector */
     TRUE, /* want to get my own publications */
     hMsgConsumer).
RUN startReceiveMessages IN hSession.
RUN waitForMessages IN hSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hSession.
MESSAGE "Number of records processed: " + STRING(numRecsRead).
PROCEDURE messageHandler:
  DEFINE INPUT PARAMETER hMessage         AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  /* hAutoReply is not used in this example */
  DEFINE OUTPUT PARAMETER hAutoReply      AS HANDLE NO-UNDO.
  DEFINE VARIABLE bh1  AS HANDLE NO-UNDO.
  DEFINE VARIABLE bh2  AS HANDLE NO-UNDO.
  DEFINE VARIABLE qh1  AS HANDLE NO-UNDO.
  DEFINE VARIABLE ttH1 AS HANDLE NO-UNDO.
  IF DYNAMIC-FUNCTION("getMessageType" in hMessage) =
    "TemptableMessage" THEN DO:
    ttH1 = DYNAMIC-FUNCTION("getTempTable" IN hMessage, ?, ?, ?).
    numRecsRead = numRecsRead + 1.
    bh1 = tth1:DEFAULT-BUFFER-HANDLE.
    CREATE QUERY qh1.
    qh1:SET-BUFFERS(bh1).
    qh1:QUERY-PREPARE("for each tcust").
    qh1:QUERY-OPEN().
    qh1:GET-FIRST.
    REPEAT WHILE NOT qh1:QUERY-OFF-END:
      bh2 = bh1:BUFFER-FIELD("name").
      MESSAGE bh2:STRING-VALUE().
      qh1:GET-NEXT.
      DELETE OBJECT bh2.
    END.
    qh1:QUERY-CLOSE.
    DELETE OBJECT ttH1.
    DELETE OBJECT qh1.
  END. /* getMessageType */
  ELSE
    stillWaiting = FALSE.
  RUN deleteMessage IN hMessage.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
  RETURN stillWaiting.
END.
```

DataSetMessage

OpenEdge applications use ProDataSets for data. The new `DataSetMessage` supports using ProDataSets for JMS messaging. The `XMLMessage` is the basis for the `DataSetMessage`. The OpenEdge Adapter for SonicMQ converts the ProDataSet to/from XML and uses the `XMLMessage` to send/receive ProDataSets. A non-OpenEdge application receives a `DataSetMessage` as an `XMLMessage`. The JMS header property signals an OpenEdge application that the incoming message is a `DataSetMessage`.

For more information on accessing the examples files, see OpenEdge messages.

For an alphabetical API reference, see ABL - JMS API Reference.

dsmsg_send.p

The following example shows sending a `DataSetMessage`.

```
DEFINE VARIABLE hds1     AS HANDLE  NO-UNDO.
DEFINE VARIABLE hMessage AS HANDLE  NO-UNDO.
DEFINE VARIABLE hSession AS HANDLE  NO-UNDO.
DEFINE VARIABLE ptp      AS LOGICAL NO-UNDO INITIAL TRUE.
DEFINE VARIABLE ret      AS LOGICAL NO-UNDO.
/* Definition for TEMP-TABLE ttCustomer */
DEFINE TEMP-TABLE ttCustomer NO-UNDO BEFORE-TABLE ttCustBef
  FIELD CustNum  LIKE Customer.CustNum
  FIELD Name     LIKE Customer.Name COLUMN-LABEL "custlab"
    XML-NODE-TYPE "Attribute"
  FIELD Country  LIKE Customer.Country
  FIELD Comments LIKE Customer.Comments FORMAT "x(40)"
  INDEX CustNum  IS PRIMARY UNIQUE CustNum
  INDEX Name     Name
  INDEX Comments IS WORD-INDEX Comments.
/* Definition for TEMP-TABLE ttOrder */
DEFINE TEMP-TABLE ttOrder NO-UNDO
  FIELD OrderNum  LIKE Order.OrderNum
  FIELD CustNum   LIKE Order.CustNum
  FIELD OrderDate LIKE Order.OrderDate
  INDEX OrderNum  IS PRIMARY UNIQUE OrderNum
  INDEX CustOrder IS UNIQUE CustNum OrderNum
  INDEX OrderDate OrderDate.
DEFINE DATASET myds
  NAMESPACE-URI "urn:myds" NAMESPACE-PREFIX "ds"
    FOR ttCustomer, ttOrder
  DATA-RELATION custOrd FOR ttCustomer, ttOrder
    REPOSITION RELATION-FIELDS (CustNum, CustNum).
/* Creates a session object. */
IF ptp THEN
  RUN jms/ptpsession.p PERSISTENT SET hSession ("-SMQConnect").
ELSE
  RUN jms/pubsubsession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
FOR EACH Customer NO-LOCK WHERE Customer.CustNum < 4:
  CREATE ttCustomer.
  BUFFER-COPY Customer TO ttCustomer.
  FOR EACH Order OF Customer NO-LOCK:
    CREATE ttOrder.
    BUFFER-COPY Order TO ttOrder.
  END. /* FOR EACH Order */
END. /* FOR EACH Customer */
hds1 = DATASET myds:HANDLE.
/* Uncomment to write XML to a file
ret = hds1:WRITE-XML("file", "wdsCustOrd.xml", YES, ?,?,YES).
*/
RUN createDatasetMessage in hSession (OUTPUT hMessage).
RUN setDataSet IN hMessage( hds1, ?, TRUE).
IF ptp THEN
  RUN sendToQueue IN hSession ("SampleQ1", hMessage, ?, ?, ?).
ELSE
  RUN PUBLISH IN hSession ("TestTopic", hMessage, ?, ?, ?).
RUN deleteMessage IN hMessage.
RUN deleteSession in hSession.
```

dsmsg_recv.p

The following example shows receiving a `DataSetMessage`.

```
DEFINE VARIABLE hBuf         AS HANDLE  NO-UNDO.
DEFINE VARIABLE hds2         AS HANDLE  NO-UNDO.
DEFINE VARIABLE hMesg        AS HANDLE  NO-UNDO.
DEFINE VARIABLE hMsgConsumer AS HANDLE  NO-UNDO.
DEFINE VARIABLE hq           AS HANDLE  NO-UNDO.
DEFINE VARIABLE hrel         AS HANDLE  NO-UNDO.
DEFINE VARIABLE hSession     AS HANDLE  NO-UNDO.
DEFINE VARIABLE ix           AS INTEGER NO-UNDO.
DEFINE VARIABLE jx           AS INTEGER NO-UNDO.
DEFINE VARIABLE numRecsRead  AS INTEGER NO-UNDO.
DEFINE VARIABLE ptp          AS LOGICAL NO-UNDO INITIAL TRUE.
DEFINE VARIABLE ret          AS LOGICAL NO-UNDO.
DEFINE VARIABLE stillWaiting AS LOGICAL NO-UNDO INITIAL TRUE.
IF ptp THEN
  RUN jms/ptpsession.p PERSISTENT SET hSession ("-SMQConnect").
ELSE
  RUN jms/pubsubsession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setBrokerURL IN hSession ("localhost:2506").
RUN beginSession IN hSession.
RUN createMessageConsumer IN hSession
  (THIS-PROCEDURE,    /* this procedure will handle it */
   "messageHandler",  /* name of internal procedure */
    OUTPUT hMsgConsumer).
IF ptp THEN
  RUN receiveFromQueue IN hSession ("SampleQ1", ?, hMsgConsumer).
ELSE
  RUN subscribe IN hSession
    ("TestTopic",
     ?,    /* durable subscription */
     ?,    /* no message selector */
     TRUE, /* want to get my own publications */
     hMsgConsumer).
RUN startReceiveMessages IN hSession.
RUN waitForMessages IN hSession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN hSession.
MESSAGE "Number of records processed: " + STRING(numRecsRead).
PROCEDURE messageHandler:
  DEFINE INPUT PARAMETER hMessage         AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  /* hAutoReply is not used in this example */
  DEFINE OUTPUT PARAMETER hAutoReply      AS HANDLE NO-UNDO.
  DEFINE VARIABLE ttH1 AS HANDLE NO-UNDO.
  DEFINE VARIABLE bh1  AS HANDLE NO-UNDO.
  DEFINE VARIABLE bh2  AS HANDLE NO-UNDO.
  DEFINE VARIABLE qh1  AS HANDLE NO-UNDO.
  IF DYNAMIC-FUNCTION("getMessageType" in hMessage) =
    "DatasetMessage" THEN DO:
    ASSIGN
      hds2        = DYNAMIC-FUNCTION("getDataset" IN hMessage, ?, ?, ?)
      numRecsRead = numRecsRead + 1.
    MESSAGE
      "num-buffers: " hds2:NUM-BUFFERS "name: " hds2:NAME SKIP
      "nspace-info: " hds2:NAMESPACE-URI hds2:NAMESPACE-PREFIX SKIP
      "num-relations: " hds2:NUM-RELATIONS
      VIEW-AS ALERT-BOX.
    DO ix = 1 TO hds2:NUM-RELATIONS:
      hrel = hds2:GET-RELATION(ix).
      MESSAGE
        "rel name: " hrel:NAME SKIP
        "reposition: " hrel:REPOSITION SKIP
        "nested: " hrel:NESTED SKIP
        "where-str: " hrel:WHERE-STRING SKIP
        "parent: " hrel:PARENT-BUFFER:NAME SKIP
        "child: " hrel:CHILD-BUFFER:NAME SKIP
```

```
        "rel-fields: " hrel:RELATION-FIELDS
        VIEW-AS ALERT-BOX.
    END.
    DO jx = 1 TO hds2:NUM-BUFFERS:
      hBuf = hds2:GET-BUFFER-HANDLE(jx).
      MESSAGE "buf name: " hBuf:NAME VIEW-AS ALERT-BOX.
      DO ix = 1 TO hBuf:NUM-FIELDS:
        MESSAGE
          "name:        " hBuf:BUFFER-FIELD(ix):NAME skip
          "type:        " hBuf:BUFFER-FIELD(ix):DATA-TYPE skip
          "extent:      " hBuf:BUFFER-FIELD(ix):EXTENT skip
          "decimals:    " hBuf:BUFFER-FIELD(ix):DECIMALS skip
          "xmltype:     " hBuf:BUFFER-FIELD(ix):XML-DATA-TYPE skip
          "xmlnodetype: " hBuf:BUFFER-FIELD(ix):XML-NODE-TYPE skip
          "initial:     " hBuf:BUFFER-FIELD(ix):INITIAL skip
          "format:      " hBuf:BUFFER-FIELD(ix):FORMAT skip
          "help:        " hBuf:BUFFER-FIELD(ix):HELP skip
          VIEW-AS ALERT-BOX.
      END.
    END.
    CREATE QUERY hq.

    DO jx = 1 TO hds2:NUM-BUFFERS:
      hBuf = hds2:GET-BUFFER-HANDLE(jx).
      MESSAGE "buf name: " hBuf:NAME VIEW-AS ALERT-BOX.
      hq:SET-BUFFERS(hBuf).
      hq:QUERY-PREPARE("FOR EACH " + hBuf:NAME).
      hq:QUERY-OPEN.

      DO WHILE hq:GET-NEXT():
        DO ix = 1 TO hBuf:NUM-FIELDS:
          MESSAGE
            "name:        " hBuf:BUFFER-FIELD(ix):NAME SKIP
            "value:       "
            IF  hBuf:BUFFER-FIELD(ix):EXTENT > 0 THEN
              hBuf:BUFFER-FIELD(ix):BUFFER-VALUE[1]
            ELSE
              hBuf:BUFFER-FIELD(ix):BUFFER-VALUE
            VIEW-AS ALERT-BOX.
        END.
      END.
    END.
    DELETE OBJECT hq.
    DELETE OBJECT hds2.
  END.
  ELSE
    stillWaiting = FALSE.
  RUN deleteMessage IN hMessage.
END PROCEDURE.
FUNCTION inWait RETURNS LOGICAL:
  RETURN stillWaiting.
END.
```

5

Configure JMS-provider proprietary features

Provider-specific proprietary APIs can be accessed by the Progress OpenEdge JMS Adapter using JNDI to find JMS-Administered Objects. Provider-specific proprietary APIs include configuring fault tolerance, load balancing, and message failover. For information on how to configure JNDI to find JMS-Administered Objects, see Build the AdminObjectFinder for use with JNDI on page 42.

This chapter details how to configure fault tolerance and other JMS-provider specific features for multiple JMS providers.

For details, see the following topics:

- Configure fault tolerance and high availability for ActiveMQ Artemis
- Access JMS Administered Objects for SonicMQ
- Configure client persistence for SonicMQ
- Configure fault tolerance for SonicMQ
- Load balance SonicMQ brokers
- Manage fail-over support for SonicMQ
- Configure message selectors for SonicMQ

Configure fault tolerance and high availability for ActiveMQ Artemis

This topic describes how to connect to ActiveMQ Artemis using ClientConnect mode through JNDI to achieve fault tolerance and high availability.

Fault tolerance and high availability can be achieved by creating an ActiveMQ Artemis cluster. Fault tolerance refers to creating redundancy in terms of software, so this example involves creating a cluster of ActiveMQ Artemis brokers. High availability refers to ensuring hardware doesn't fail. To achieve high availability, create the ActiveMQ Artemis brokers on different machines.

Prerequisites

This topic assumes you have the ActiveMQ Artemis binaries, and that the Progress OpenEdge JMS Adapter is installed and configured on your machine.

Main steps

To achieve a fault-tolerant, highly available ActiveMQ Artemis cluster configuration, follow these steps:

1. Navigate to the location where you want to create ActiveMQ Artemis master and worker brokers. For example: `C:\JMS\ArtemisMQ`.
2. Run the following `artemis create` commands to create master and worker brokers. For example:

```
C:\JMS\ArtemisMQ>apache-artemis-2.11.0-bin\apache-artemis-2.11.0\bin\artemis create
 master --http-port 8162 --default-port 81617

C:\JMS\ArtemisMQ>apache-artemis-2.11.0-bin\apache-artemis-2.11.0\bin\artemis create
 worker --http-port 8163 --default-port 81618
```

Note: Because the above commands create ActiveMQ Artemis brokers on localhost, provide admin/admin for username and password, and 'N' for anonymous access. To achieve high availability, create these brokers on different machines.

These commands create `master` and `worker` directories in the location where they were run.

3. In the `master\etc` directory, open the `broker.xml` file and locate the `<acceptors>` tag.
4. Remove all content between the `<acceptors>` opening and closing tag, including the `<acceptors>` tags themselves. Replace that content with the following information:

```
<connectors>
 <connector name="artemis">tcp://localhost:61617</connector>
</connectors>

<ha-policy>
 <replication>
  <master>
  <check-for-live-server>true</check-for-live-server>
  </master>
 </replication>
</ha-policy>

<acceptors>
 <acceptor name="artemis">tcp://localhost:61617</acceptor>
</acceptors>

<cluster-user>cluster</cluster-user>
<cluster-password>cluster</cluster-password>

<broadcast-groups>
 <broadcast-group name="bg-group1">
  <group-address>231.7.7.7</group-address>
  <group-port>9876</group-port>
  <broadcast-period>5000</broadcast-period>
  <connector-ref>artemis</connector-ref>
 </broadcast-group>
</broadcast-groups>
<discovery-groups>
 <discovery-group name="dg-group1">
  <group-address>231.7.7.7</group-address>
  <group-port>9876</group-port>
  <refresh-timeout>10000</refresh-timeout>
 </discovery-group>
</discovery-groups>

<cluster-connections>
 <cluster-connection name="my-cluster">
  <connector-ref>artemis</connector-ref>
  <message-load-balancing>ON_DEMAND</message-load-balancing>
  <max-hops>0</max-hops>
  <discovery-group-ref discovery-group-name="dg-group1"/>
 </cluster-connection>
</cluster-connections>
```

5. In the `worker\etc` directory, open the `broker.xml` file and locate the `<acceptors>` tag.

6. Remove all content between the <acceptors> opening and closing tag, including the <acceptors> tags themselves. Replace that content with the following information:

```
<ha-policy>
 <replication>
  <slave>
   <allow-failback>true</allow-failback>
   <failback-delay>5000</failback-delay>
  </slave>
 </replication>
</ha-policy>

<connectors>
 <connector name="artemis">tcp://localhost:61618</connector>
 <connector name="netty-live-connector">tcp://localhost:61617</connector>
</connectors>

<acceptors>
 <acceptor name="artemis">tcp://localhost:61618 </acceptor>
</acceptors>

<cluster-user>cluster</cluster-user>
<cluster-password>cluster</cluster-password>

<broadcast-groups>
 <broadcast-group name="bg-group1">
  <group-address>231.7.7.7</group-address>
  <group-port>9876</group-port>
  <broadcast-period>5000</broadcast-period>
  <connector-ref>artemis</connector-ref>
 </broadcast-group>
</broadcast-groups>
<discovery-groups>
 <discovery-group name="dg-group1">
  <group-address>231.7.7.7</group-address>
  <group-port>9876</group-port>
  <refresh-timeout>10000</refresh-timeout>
 </discovery-group>
</discovery-groups>

<cluster-connections>
 <cluster-connection name="my-cluster">
  <connector-ref>artemis</connector-ref>
  <message-load-balancing>ON_DEMAND</message-load-balancing>
  <max-hops>0</max-hops>
  <discovery-group-ref discovery-group-name="dg-group1"/>
 </cluster-connection>
</cluster-connections>
```

7. Navigate to the master\bin directory and run the following command:

```
C:\JMS\ArtemisMQ\master\bin>artemis run
```

8. In a new terminal window, navigate to the worker\bin directory and run the following command:

```
C:\JMS\ArtemisMQ\worker\bin>artemis run
```

When you see the “backup announced” text in the worker console that means the cluster connection was created.

9. In the `C:\JMS\ArtemisMQ` directory, create an `AdminObjectFinder.Java` and a `jndi.properties` file. Code for creating the `AdminObjectFinder.java` and `jndi.properties` files for ActiveMQ Artemis can be found in the AdminObjectFinder.java and jndi.properties example for ActiveMQ Artemis on page 45 topic.
10. Copy the `artemis-jms-client-all-`*`version`*`.jar` file into the `C:\JMS\ArtemisMQ` directory. For example:

```
C:\JMS\ArtemisMQ>copy
apache-artemis-2.11.0-bin\apache-artemis-2.11.0\lib\client\artemis-jms-client-all-2.11.0.jar
 .
```

11. Run the following command to compile the `AdminObjectFinder.java` file and generate a `jmsfromABL` package:

```
C:\JMS\ArtemisMQ>javac -cp artemis-jms-client-all-2.11.0.jar;.; -d .
AdminObjectFinder.java
```

12. Run the following command to generate an `AdminObjectFinder.jar` file:

```
C:\JMS\ArtemisMQ>jar -cvf AdminObjectFinder.jar jmsfromABL jndi.properties
```

13. Add the `AdminObjectFinder.jar` file to the classpath in `$DLC/properties/JavaTools.properties` along with the ActiveMQ Artemis client jars. For example:

```
[SonicMQ]
[illegible]
jvmargs=-DsonicMQExtensions=false -DjmsProvider=ArtemisMQ
```

14. Create and run the following `Producer_JNDI.p` file in prowin:

```
DEFINE VARIABLE ptpsession AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.
DEFINE VARIABLE lDebug AS LOGICAL.
DEFINE VARIABLE cdate AS CHARACTER NO-UNDO FORMAT "x(16)".
DEFINE VARIABLE ddate AS DATE      NO-UNDO.


/* Creates a session object. */
RUN jms/ptpsession.p PERSISTENT SET ptpsession ("-SMQConnect").

/* Set user credentials. */
RUN setBrokerURL      IN ptpsession (INPUT "ConnectionFactory").
RUN setUser           IN ptpsession (INPUT "admin").
RUN setPassword       IN ptpsession (INPUT "admin").

/* Connect to the broker. */
RUN beginSession IN ptpsession.

/* Create a text message */
RUN createTextMessage IN ptpsession (OUTPUT messageH).
ddate = DATE(cdate).
RUN setText IN messageH ("This is test message sent.").

/* Publish the message on the "RequestQ" topic */
RUN sendToQueue IN ptpsession ("example.MyQueue", messageH, ?, ?, ?).
RUN deleteMessage IN messageH.
RUN deleteSession IN ptpsession.

Message "Sent".
```

Access JMS Administered Objects for SonicMQ

Client applications can dynamically adjust to redefinition of the broker connections and the destinations where messages are sent and received. This is achieved when client applications look up connection information in serialized connection objects or a store of JMS administered objects. You can access JMS Administered Objects for SonicMQ either through a file system using Serialized Connection Objects or through the Sonic server using the JNDI Naming Service.

The following topics describe:

- Create a serialized connection object on page 111
- Find JMS Administered Objects in JNDI or proprietary directories on page 115

Create a serialized connection object

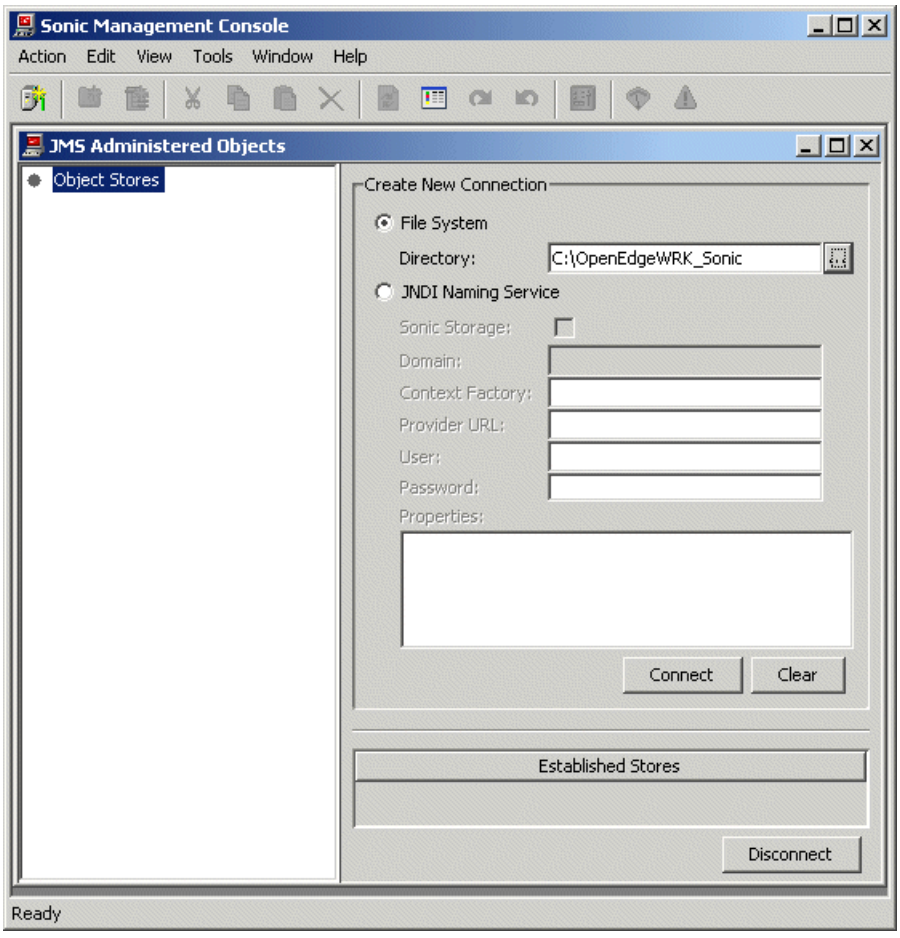
A *serialized connection object* contains all the connection information required by a client to connect to a SonicMQ Broker, including `userid` and `password`. A SonicMQ administrator creates the serialized connection object as a file using the Sonic Management Console and provides the serialized connection object to the OpenEdge client. The OpenEdge client uses the setConnectionFile procedure on page 158 with the file containing the serialized connection object when creating the messaging session.

The serialized connection object file is used when connecting to a SonicMQ Broker. The following example shows how to use the serialized connection object file `MyConnectionObject.sjo`:

```
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setConnectionFile IN hSession ("MyConnectionObject.sjo").
RUN beginSession IN hSession.
```

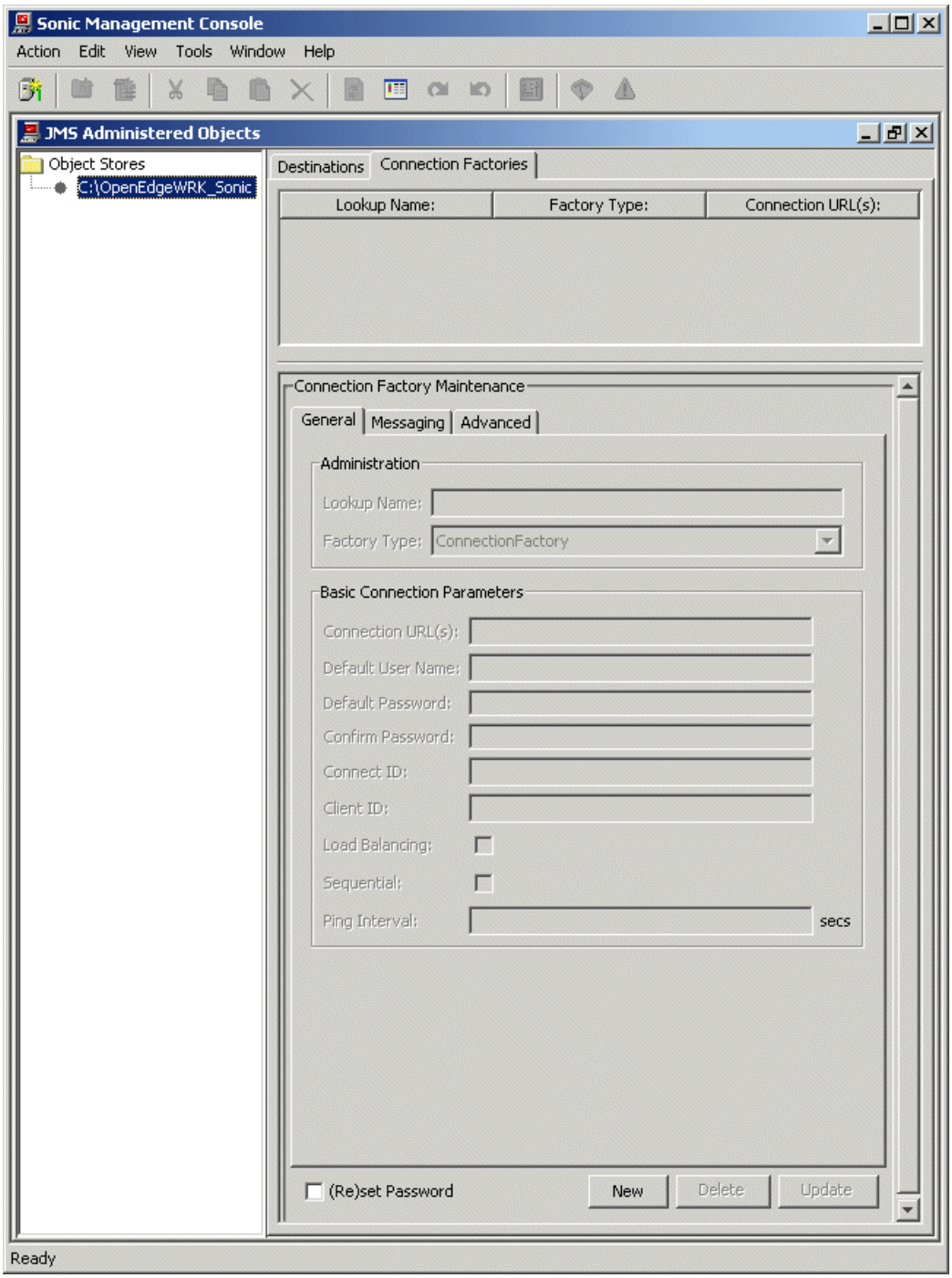
To create a serialized connection object file, follow these steps:

1. Ensure the Domain Manager for SonicMQ is running.
2. Ensure the Sonic Management Console is started and connected to the Domain Manager.
3. On the Sonic Management Console menu bar, select **Tools** and then **JMS Administered objects**.
4. Select the **File System** and navigate to the **Directory** you want the serialized connection object file to reside, as shown:

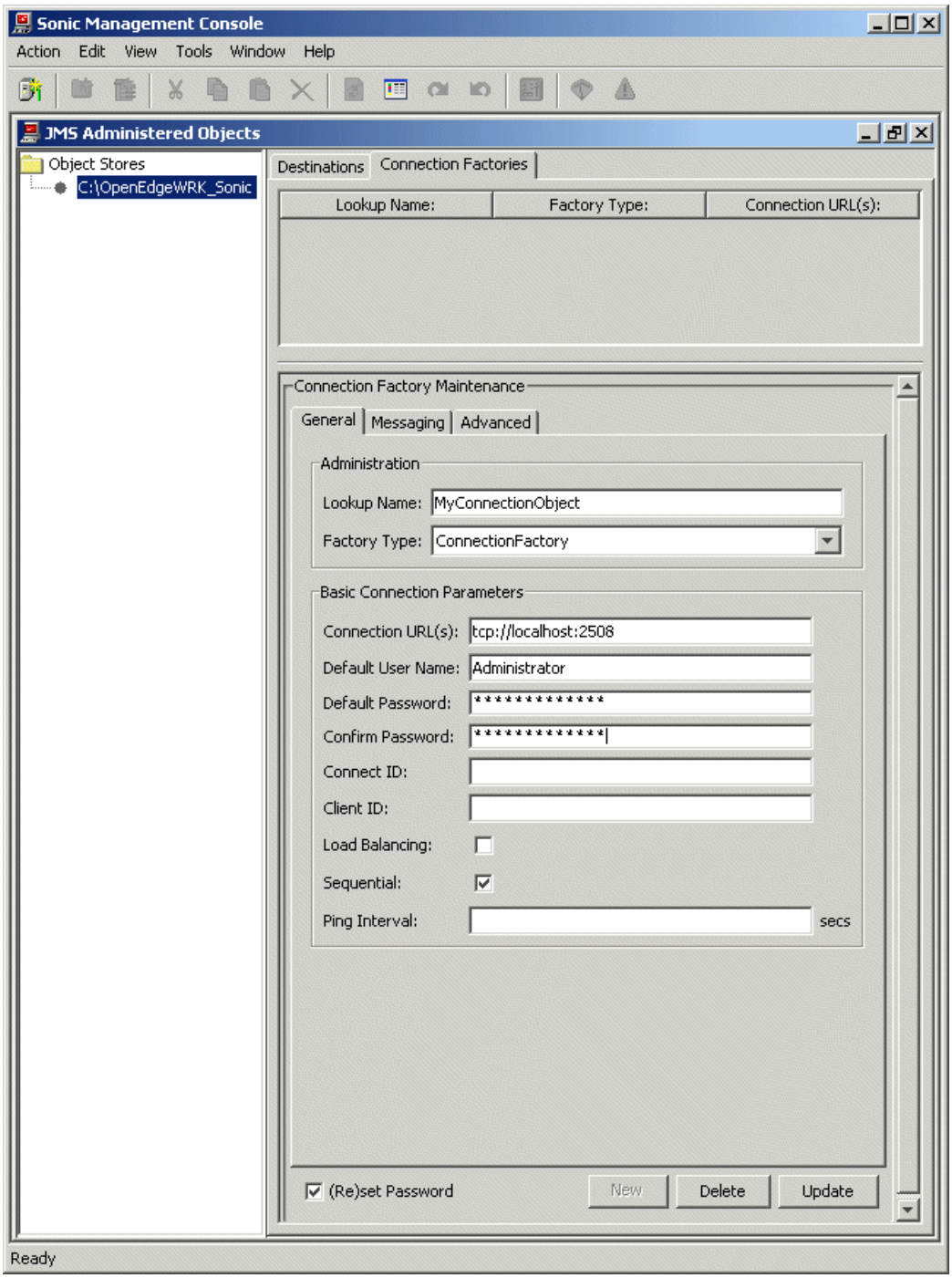


5. Select the **Connect** button.

6. In the left pane, select the Object Store directory that you created and then select the **Connection Factories** tab, as shown:

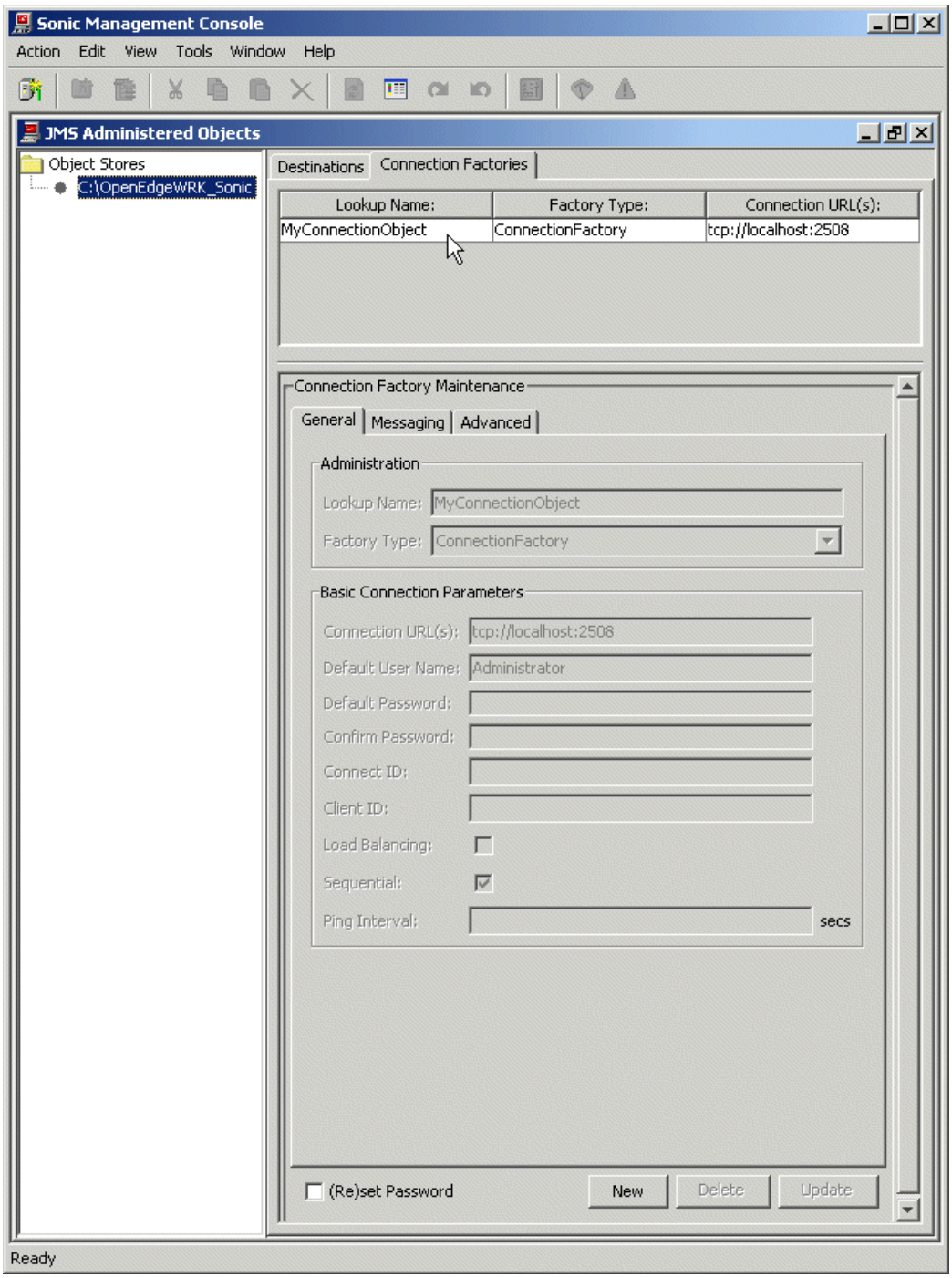


7. Select the **New** button and enter the required connection information for **Lookup Name** (name of serialized connection object file), **Connection URL(s)**, **Default User Name**, **Default Password**, and **Confirm Password**, as shown:



Note: All other information is optional for the connection object.

8. Select **Update**. The serialized connection object appears an entry, as shown:



The serialized connection object file `MyConnectionObject.sjo` exists in the Object Stores specified directory. An OpenEdge client uses the file `MyConnectionObject.sjo` to connect to the SonicMQ Broker.

You can access the `MyConnectionObject.sjo` file in an a procedure:

```
define variable sessionHandler as handle.
define variable messageHandler as handle.

/* Set up connect procedure persistantly on the JMS adapter */
/* BrokerConnect connection string: */
/* run jms/jmssession.p persistent set sessionHandler("-H localhost -S 3620
-DirectConnect"). */
/* ClientConnect connection string: */
RUN jms/jmssession.p persistent set sessionHandler("-SMQConnect").

/* connect using Serialized Connection Object: */
RUN setConnectionFile IN sessionHandler("C:\OpenEdgeWRK_Sonic\MyConnectionObject.sjo").

RUN beginSession in sessionHandler.
MESSAGE "Connected to Sonic" VIEW-AS ALERT-BOX.

/* start using ABL JMS APIs */
/* Create a message */
RUN createTextMessage in sessionHandler(output messageHandler).
RUN setText in messageHandler("A message from the man himself. I am the best.").

/* Send the message to the "queue" using jmssession or ptp, pubsub is only topics */
/* Ensure the queue name exists as a destination */
RUN sendToQueue in sessionHandler("SampleQ1", messageHandler, 0, 0, "PERSISTENT").
MESSAGE "Message Sent to Sonic" VIEW-AS ALERT-BOX.

/* Delete message and session */
RUN deleteMessage in messageHandler.
RUN deleteSession in sessionHandler.
```

Find JMS Administered Objects in JNDI or proprietary directories

A JMS Administered Object is an object created by a JMS administrator and registered with a directory (typically a JNDI-compliant directory) under a name that is meaningful to the JMS clients. The object contains JMS configuration information that is created by a JMS administrator and later used by JMS clients. Java Naming and Directory Interface (JNDI) is an interface for JMS administrators to create and configure administered objects and store them in a namespace.

The SonicMQ-administered objects are:

- `TopicConnectionFactory`
- `QueueConnectionFactory`
- `Topic`
- `Queue`

For example, the administrator creates a `TopicConnectionFactory` object, which contains all the JMS server connection parameters (communication protocol host and port), assigns it a name, and stores it in a JNDI directory. The client does not have to know the connection parameters to connect to the JMS server. The client finds the object by name in the directory and uses it to create connection objects. The administrator can change the connection parameters later without affecting client applications.

The administrator can give the `Topic` and `Queue` objects meaningful aliases to shield the client from their internal names. For example, a topic with the internal JMS name of `sports.USA.Northeast.golf` could be stored in the directory under `northern.golfers`. For more information on administered objects, see the *Java Message Service* specification, *SonicMQ Programming Guide*,and *SonicMQ Configuration and Management Guide*.

jmsfrom4gl.AdminObjectFinder class

The following code is the skeleton of the `jmsfrom4gl.AdminObjectFinder` class. Use it as a template to create a class file and install it on the OpenEdge Adapter for the SonicMQ host; Unified Broker host for BrokerConnect, OpenEdge client host for ClientConnect, and AppServer or WebSpeed Transaction server host for the ServerConnect option.

The `jmsfrom4gl.AdminObjectFinder` name is mandatory. The class and the `get...()` methods must be declared public. The `AdminObjectFinder` class must be part of the `jmsfrom4gl` package and placed in a directory called `jmsfrom4gl`. The directory that contains `jmsfrom4gl` must be on the `CLASSPATH` of the OpenEdge Adapter for SonicMQ.

For example:

```
package jmsfrom4gl;
import javax.jms.TopicConnectionFactory;
import javax.jms.QueueConnectionFactory;
import javax.jms.Topic;
import javax.jms.Queue;
public class AdminObjectFinder
{  public TopicConnectionFactory getTopicConnectionFactory(String name)
    throws Exception
  {
    TopicConnectionFactory factory = null;
    // Write code to populate factory
    return factory;
  }
  public QueueConnectionFactory getQueueConnectionFactory(String name)
    throws Exception
  {
    QueueConnectionFactory factory = null;
    // Write code to populate factory
    return factory;
  }
  public Topic getTopic(String name)
    throws Exception
  {
    Topic topic = null;
    // Write code to populate topic
    return topic;
  }
  public Queue getQueue(String name)
    throws Exception
  {
    Queue queue = null;
    // Write code to populate queue
    return queue;
  }
```

Note: The *brokerURL* startup parameter is used as the input parameter for the `getTopicConnectionFactory` and `getQueueConnectionFactory` methods. For example, if the OpenEdge application calls the setBrokerURL procedure on page 247 passing in the input parameter *directory_factory_name*, the ABL–JMS implementation on the server side calls the `getTopicConnectionFactory` method with *directory_factory_name* as the parameter. If the `getTopicConnectionFactory` and `getQueueConnectionFactory` methods are implemented, the `jmsServerName` startup parameter is ignored (since the identity of the server's vendor is encapsulated in the object). It is sufficient to implement methods for those objects that should be obtained from the directory. For example, it is legal to have an `AdminObjectFinder` class with only the `getTopicConnectionFactory` method. The ABL–JMS implementation looks for the methods dynamically and does not fail if the other methods are missing. If the object finder method returns null, the ABL–JMS implementation tries to create the object as if the method is not there. For more information on setting the `CLASSPATH`, see Configure your JMS provider on page 39.

Connection file parameters

A connection file allows you to configure a set of connection parameters. The following table lists the connection parameters that can be set for each connection type.

Table 21: Connection file parameters (Continued)

Parameter	Connection type
Broker-side selectors	All[1]
Client ID	All
Connection URL	All
Connect ID	All
Default password	All
Default user name	All
Durable message order	All
Flow to disk	All
Load Balancing	All
Max delivery count	All
Persistent delivery mode	All
Ping interval	All
Prefetch count	All
Prefetch threshold	All

[1] All connections include BrokerConnect and ClientConnect

Parameter	Connection type
Sequential	All
Client transaction buffer size	Client-connect
Enable fault tolerant	Client-connect
Enable local store	Client-connect
Fault tolerant reconnect timeout	Client-connect
Initial connect timeout	Client-connect
Local store directory	Client-connect
Local store size	Client-connect
Reconnect interval	Client-connect
Reconnect timeout	Client-connect

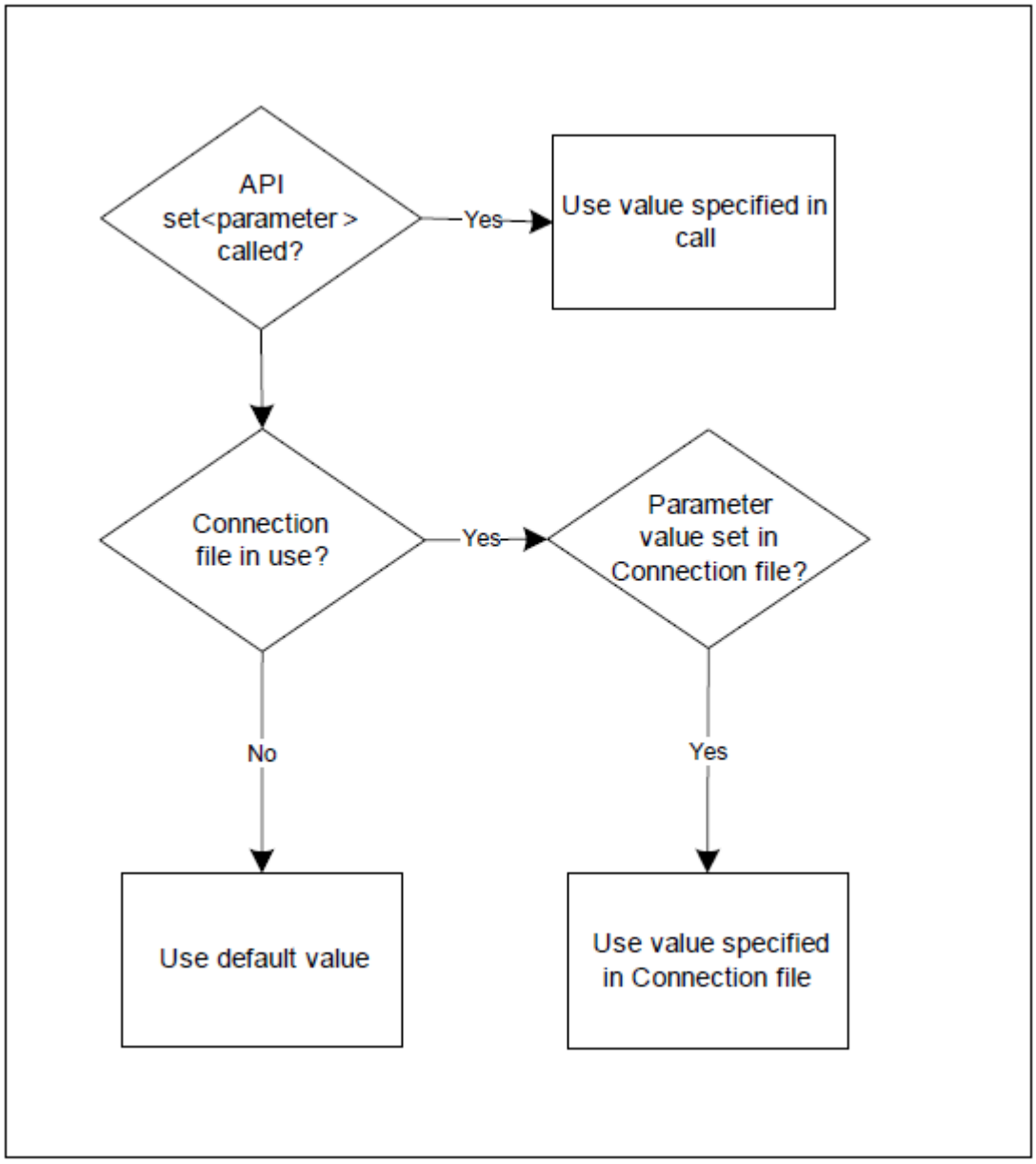
Note: The combined length of the Client ID, Default password, and other connection file parameters listed in the table above must not exceed 30,000 non-Unicode characters when connecting to an application server.

Parameter values are fixed when the beginSession procedure on page 172 is called. When the beginSession procedure on page 172 is called, the following process takes place for each parameter to determine its value:

1. If the parameter value was set by a call to the corresponding `set<parameter>`routine, the values set in the call is used. If the same `set<parameter>`routine is called more than once, the last setting is used.
2. If the parameter value was not set by a `set<parameter>`routine, and a parameter file is specified, and the parameter is set in the connection file, then the value in the connection file is used.
3. If the parameter is not set by a `set<parameter>`routine, and it is not set in a specified connection file, then the default value is used.

The following figure depicts the process for determining the parameter value for each parameter during execution of the beginSession procedure on page 172.

Figure 1: Connection parameter value decision flow



All connection parameters must be set prior to calling `beginSession`.

Configure client persistence for SonicMQ

Client persistence provides a higher level of reliability than is defined in the JMS specification. *Client persistence* allows the JMS session to continue sending messages regardless of the SonicMQ Broker status. If the SonicMQ Broker is not available, the messages are stored locally and sent when the SonicMQ Broker becomes available.

Note: Client persistence is only available to OpenEdge clients running in ClientConnect mode.

Storing undeliverable messages

When the connection to the SonicMQ Broker fails, messages are persisted to disk, and replayed when the connection is re-established. Each connection must have a local directory specified where messages will be stored when a connection fails.

The following table lists the methods for managing client persistence.

Table 22: Managing client persistence

Setting	Getting
setLocalStoreDirectory procedure on page 162 setLocalStoreSize procedure on page 163 setReconnectTimeout procedure on page 167 setReconnectInterval procedure on page 167 setClientPersistence procedure on page 156 createRejectedMessageConsumer procedure on page 144	getLocalStoreDirectory function on page 151 getLocalStoreSize function on page 151 getReconnectTimeout function on page 153 getReconnectInterval function on page 152 getClientPersistence function on page 146

Additionally, client persistence requires using the setClientID procedure on page 250. The `clientID` must be unique for each client. The application may optionally call the setPingInterval procedure on page 164 to enable connection checking between the client and the SonicMQ Broker.

Note: Creating a Rejected Message Listener is also optional. This listener notifies the client when a message is rejected during playback.

The caller must ensure that the connections to the machine and port number are correct. It is possible for messages to be lost if an incorrect broker is specified. Although the messages will be persisted to disk, they will never be sent since there will never be a broker to connect to.

Note: Client persistence does not support Message Consumers and transacted sessions.

Client persistence example for SonicMQ

Client persistence allows the JMS session to continue sending messages regardless of the SonicMQ Broker status. If the SonicMQ Broker is not available, the messages are stored locally and sent when the SonicMQ Broker becomes available.

For more information on client persistence, see Configure client persistence for SonicMQ on page 119.

The following code sample shows how to set up client persistence.

Client persistence example

```
DEFINE VARIABLE hSession     AS HANDLE NO-UNDO.
DEFINE VARIABLE rejectedMsgH AS HANDLE NO-UNDO.

/* Run adapter as symbiotic process */
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").

/* Set local store directory off of current working directory */
RUN setLocalStoreDirectory IN hSession ("mqstore").
RUN setLocalStoreSize IN hSession (5000).

/* Set timeouts - Retry every 5 minutes and give up if broker down 10 hours */
RUN setReconnectTimeout IN hSession (600).
RUN setReconnectInterval IN hSession (300).
RUN setClientPersistence IN hSession (TRUE).
RUN setBrokerURL IN hSession ("MQBrokerHost:2506").
RUN setClientID IN hSession ("SomeUniqueName").
RUN beginSession IN hSession.

/* Once session is established, create rejected Message Consumer */
RUN createRejectedMessageConsumer IN hSession
  (THIS-PROCEDURE, "RejectedMsgHandler", OUTPUT rejectedMsgH).

PROCEDURE RejectedMsgHandler:
  DEFINE INPUT PARAMETER hMessage         AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMessageConsumer AS HANDLE NO-UNDO.
  /* hAutoReply is not used in this example */
  DEFINE OUTPUT PARAMETER hAutoReply      AS HANDLE NO-UNDO.
  DEFINE VARIABLE contentType   AS CHARACTER NO-UNDO.
  DEFINE VARIABLE errorCode     AS CHARACTER NO-UNDO.
  DEFINE VARIABLE errorText     AS CHARACTER NO-UNDO.
  DEFINE VARIABLE exceptionCode AS CHARACTER NO-UNDO.
  DEFINE VARIABLE hMessagePart  AS HANDLE    NO-UNDO.
  DEFINE VARIABLE iNumParts     AS INTEGER   NO-UNDO.
  DEFINE VARIABLE msgType       AS CHARACTER NO-UNDO.
  MESSAGE "Reject message" VIEW-AS ALERT-BOX.
  errorCode = DYNAMIC-FUNCTION("getCharProperty" IN hMessage, "errorCode").
  errorText = DYNAMIC-FUNCTION("getCharProperty" IN hMessage, "errorText").
  exceptionCode = DYNAMIC-FUNCTION("getCharProperty" IN hMessage,
    "exception").
  MESSAGE errorText VIEW-AS ALERT-BOX.
  DISPLAY errorCode exceptionCode.
  iNumParts = DYNAMIC-FUNCTION("getPartCount" IN hMessage).
  IF DYNAMIC-FUNCTION("isMessagePart" IN hMessage, 1) = TRUE THEN DO:
    contentType = DYNAMIC-FUNCTION("getMessagePartByIndex":U IN hMessage,
                    INPUT iNumParts, OUTPUT hMessagePart).
    msgType     = DYNAMIC-FUNCTION('getMessageType':U IN hMessagePart).
    DISPLAY iNumParts msgType contentType.
  END.
  RUN deleteMessage IN hMessage.
END PROCEDURE.
```

Configure fault tolerance for SonicMQ

Fault tolerant connections allow a JMS client to reconnect to a SonicMQ Broker and enable reconnection to the same SonicMQ Broker, or to one of the SonicMQ Broker specified in a list, if this has been defined before the session is created. Fault tolerance is set on the client but must be supported by the SonicMQ Broker.

Note: Fault tolerance is only available to OpenEdge clients running in ClientConnect and ServerConnect mode.

An OpenEdge client specifies participation in a fault tolerant session when the client connects to licensed fault tolerant SonicMQ Brokers. In a fault tolerant session, when the SonicMQ Broker or the network experiences a fault, the session resumes when the SonicMQ Broker or its backup is available. The client maintains connection and session information waiting for the SonicMQ Broker to be available.

Replicated SonicMQ Brokers

Replicated SonicMQ Brokers provide additional broker availability. The active SonicMQ Broker and the replicated SonicMQ Broker synchronize all client information and data. If the active SonicMQ Broker goes down, the replicated SonicMQ Broker takes over as the lead broker. Clients running a fault tolerant connection seamlessly connect to the replicated SonicMQ Broker.

The following table lists the methods for managing fault tolerant connections.

Table 23: Managing fault tolerance

Setting	Getting
setFaultTolerant procedure on page 159 setClientTransactionBufferSize procedure on page 157 setInitialConnectionTimeout procedure on page 161 setFaultTolerantReconnectTimeout procedure on page 160 createChangeStateConsumer procedure on page 143	isFaultTolerant function on page 155 getFaultTolerant function on page 148 getClientTransactionBufferSize function on page 147 getInitialConnectionTimeout function on page 150 getFaultTolerantReconnectTimeout function on page 149

Fault tolerance examples for SonicMQ

Fault tolerant connections allow another SonicMQ Broker to take over if the original SonicMQ Broker fails. To ensure message delivery, use the fault-tolerant APIs to setup and enable fault tolerance. These APIs include the setFaultTolerant procedure on page 159, the getFaultTolerant function on page 148, the isFaultTolerant function on page 155, the setConnectionURLs procedure on page 158, the setFaultTolerantReconnectTimeout procedure on page 160, the getFaultTolerantReconnectTimeout function on page 149, the setInitialConnectionTimeout procedure on page 161, the getInitialConnectionTimeout function on page 150, the setClientTransactionBufferSize procedure on page 157, the getClientTransactionBufferSize function on page 147, and the createChangeStateConsumer procedure on page 143. Although you setup and enable fault tolerance from the SonicMQ client, the SonicMQ Broker must support it.

Note: Fault tolerance is only available to OpenEdge clients running in ClientConnect and ServerConnect.

After creating the session object, you must create the list of SonicMQ Brokers to use, set the fault tolerant property for the session, and then start the session.

Fault tolerance set up

The following example shows how to set up a fault tolerant session.

```
DEFINE VARIABLE hSession AS HANDLE NO-UNDO.
RUN jms/jmssession.p PERSISTENT SET hSession ("-SMQConnect").
RUN setConnectionURLs IN hSession ("Primary:2508,BackupServer:9876").
RUN setFaultTolerant IN hSession(TRUE).
RUN beginSession IN hSession.
```

Example of a "ChangeState" handler (optional)

When the connection to the SonicMQ Broker is lost, SonicMQ has the ability to notify the application. A special asynchronous handler, "ChangeState" handler, notifies the client application whenever the state of the SonicMQ Broker changes. The character header property of the message passed to the "ChangeState" handler contains one of the following values: `active`, `reconnecting`, `failed`, or `closed`. You setup the handler by calling the createChangeStateConsumer procedure on page 143 after to calling the beginSession procedure on page 172.

The following code sample shows how to use the createChangeStateConsumer procedure.

```
RUN createChangeStateConsumer IN hSession
  (THIS-PROCEDURE, "msgHandler", OUTPUT hMessage).
PROCEDURE msgHandler:
  DEFINE INPUT PARAMETER hMessage     AS HANDLE NO-UNDO.
  DEFINE INPUT PARAMETER hMsgConsumer AS HANDLE NO-UNDO.
  DEFINE OUTPUT PARAMETER hReply      AS HANDLE NO-UNDO.
  DEFINE VARIABLE cValue AS CHARACTER NO-UNDO.
  /* cValue will be "active", "reconnecting", "failed", or "closed" */
  cValue = DYNAMIC-FUNCTION("getCharProperty" IN hMessage, "state").
  DISPLAY cValue.
  RUN deleteMessage IN hMessage.
END PROCEDURE.
```

Load balance SonicMQ brokers

Sonic supports the creation of load-balanced clusters. By default, connect-time load balancing is enabled for all SonicMQ Brokers within a cluster. When load balancing is in effect, connection requests can be redirected to other brokers in the cluster for more efficient processing.

To manage load balancing for the current request, use the setLoadBalancing procedure on page 265 and the getLoadBalancing function on page 207.

Manage fail-over support for SonicMQ

Sonic allows a client to specify a list of Sonic brokers to connect. This makes it easier for the client to establish a connection when one or more brokers are not available. Sonic also allows the application to specify whether to try connecting to the brokers in the list sequentially or randomly. The following table lists the methods for managing broker connections.

Table 24: Managing fail-over support

Setting	Getting
setConnectionURLs procedure on page 158 setSequential procedure on page 168	getConnectionURLs function on page 147 getSequential function on page 154

Configure message selectors for SonicMQ

SonicMQ messages can be filtered so that only messages meeting a specific criteria will be received. Message consumers in both domains can apply message selectors to filter messages. *Message selectors* filter messages so a client does not receive all the messages. Message selectors evaluate message header fields and properties. They do not access the body of a message.

The default behavior of message selector filtering is:

- For PTP sessions, the filtering is always performed by the SonicMQ Broker
- For Pub/Sub sessions, all messages for a subscribed topic are by default delivered to the subscriber, then the filter is applied by the SonicMQ client to decide which messages to consume

To have the SonicMQ Broker perform the filtering for a Pub/Sub session, use the setSelectorAtBroker procedure on page 168 and getSelectorAtBroker function on page 154. Choosing to perform message selection at the SonicMQ Broker reduces message traffic between the broker and the client but increases the workload of the SonicMQ Broker.

Note: Server-based message selectors are available with all adapters.

6

ABL-JMS API Reference

This reference content details all of the supported ABL-JMS APIs for the Progress OpenEdge JMS Adapter.

This content includes topics describing the SonicMQ proprietary APIs on page 139, which are supported only when using SonicMQ as a JMS provider through the Progress OpenEdge JMS Adapter. These SonicMQ proprietary APIs were supported in the classic ABL API in the OpenEdge SonicMQ Adapter. You can still use their functionality by configuring SonicMQ as your JMS provider. For more information, see Configure your JMS provider on page 39.

For details, see the following topics:

- Session objects
- jmssession.p
- ptpsession.p
- pubsubsession.p
- Methods in the Session objects
- Methods in the Message Consumer objects
- Methods in the Message objects
- SonicMQ proprietary APIs
- addBytesPart procedure
- addTextPart procedure
- appendText procedure
- beginSession procedure

- browseQueue procedure
- cancelDurableSubscription procedure
- clearBody procedure
- clearProperties procedure
- commitReceive procedure
- commitSend procedure
- createBytesMessage procedure
- createDataSetMessage procedure
- createHeaderMessage procedure
- createMapMessage procedure
- createMessageConsumer procedure
- createStreamMessage procedure
- createTemporaryQueue procedure
- createTemporaryTopic procedure
- createTempTableMessage procedure
- createTextMessage procedure
- createXMLMessage procedure
- deleteConsumer procedure
- deleteMessage procedure
- deleteSaxWriter procedure
- deleteSession procedure
- deleteTemporaryQueue procedure
- deleteTemporaryTopic procedure
- endOfStream function
- getAdapterService function
- getApplicationContext function
- getBrokerURL function
- getBytesCount function
- getBytesToRaw function
- getChar function
- getCharCount function
- getCharProperty function
- getClientID function

- getConnectID function
- getConnectionID function
- getConnectionMetaData function
- getDataSet function
- getDate function
- getDateProperty function
- getDateTime function
- getDateTimeProperty function
- getDateTime-TZ function
- getDateTimeTzProperty function
- getDecimal function
- getDecimalProperty function
- getDefaultPersistency function
- getDefaultPriority function
- getDefaultTimeToLive function
- getDestinationName function
- getInt function
- getIntProperty function
- getInt64 function
- getInt64Property function
- getItemType function
- getJMSCorrelationID function
- getJMSCorrelationIDAsBytes function
- getJMSDeliveryMode function
- getJMSDestination function
- getJMSExpiration function
- getJMSMessageID function
- getJMSPriority function
- getJMSRedelivered function
- getJMSReplyTo function
- getJMSServerName function
- getJMSTimestamp function
- getJMSType function

- getLoadBalancing function
- getLogical function
- getLogicalProperty function
- getLongString function
- getLongStringCP function
- getLongText function
- getLongTextCP function
- getMapNames function
- getMemptr function
- getMessagePartByID function
- getMessagePartByIndex function
- getMessageType function
- getNoAcknowledge function
- getPartCount function
- getPassword
- getProcHandle function
- getProcName function
- getPropertyNames function
- getPropertyType function
- getReplyAutoDelete function
- getReplyPersistency function
- getReplyPriority function
- getReplyTimeToLive function
- getReplyToDestinationType function
- getReuseMessage function
- getSaxWriter function
- getSession function
- getShutdownWaitFor function
- getTempTable function
- getText function
- getTextPartByID function
- getTextPartByIndex function
- getTextSegment function

- getTransactedReceive function
- getTransactedSend function
- getUser function
- getX-Document function
- hasReplyTo function
- inErrorHandling function
- inMessageHandling function
- inQueueBrowsing function
- inReplyHandling function
- isMessagePart function
- JMS-MAXIMUM-MESSAGES global variable
- messageHandler procedure
- moveToNext procedure
- publish procedure
- readBytesToRaw procedure
- readChar function
- readDate function
- readDateTime function
- readDateTime-TZ function
- readDecimal function
- readInt function
- readInt64 function
- readLogical function
- readLongString function
- readLongStringCP function
- receiveFromQueue procedure
- recover procedure
- requestReply procedure
- reset procedure
- rollbackReceive procedure
- rollbackSend procedure
- sendToQueue procedure
- setAdapterService procedure

- setApplicationContext procedure
- setBoolean procedure
- setBooleanProperty procedure
- setBrokerURL procedure
- setByte procedure
- setByteProperty procedure
- setBytesFromRaw procedure
- setChar procedure
- setClientID procedure
- setDataSet procedure
- setDate procedure
- setDateProperty procedure
- setDateTime procedure
- setDateTimeProperty procedure
- setDateTime-TZ procedure
- setDateTimeTzProperty procedure
- setDefaultPersistency procedure
- setDefaultPriority procedure
- setDefaultTimeToLive procedure
- setDouble procedure
- setDoubleProperty procedure
- setErrorHandler procedure
- setFloat procedure
- setFloatProperty procedure
- setInt procedure
- setIntProperty procedure
- setInt64 procedure
- setInt64Property procedure
- setJMSCorrelationID procedure
- setJMSCorrelationIDAsBytes procedure
- setJMSReplyTo procedure
- setJMSServerName procedure
- setJMSType procedure

- setLoadBalancing procedure
- setLong procedure
- setLongProperty procedure
- setLongString procedure
- setLongText procedure
- setMemptr procedure
- setNoAcknowledge procedure
- setNoErrorDisplay procedure
- setPassword procedure
- setReplyAutoDelete procedure
- setReplyPersistency procedure
- setReplyPriority procedure
- setReplyTimeToLive procedure
- setReplyToDestinationType procedure
- setReuseMessage procedure
- setSaxReader procedure
- setShort procedure
- setShortProperty procedure
- setShutdownWaitFor procedure
- setString procedure
- setStringProperty procedure
- setTempTable procedure
- setText procedure
- setTransactedReceive procedure
- setTransactedSend procedure
- setUser procedure
- setX-Document procedure
- startReceiveMessages procedure
- stopReceiveMessages procedure
- subscribe procedure
- waitForMessages procedure
- writeBoolean procedure
- writeByte procedure

- writeBytesFromRaw procedure
- writeChar procedure
- writeDate procedure
- writeDateTime procedure
- writeDateTime-TZ procedure
- writeDouble procedure
- writeFloat procedure
- writeInt procedure
- writeInt64 procedure
- writeLong procedure
- writeLongString procedure
- writeShort procedure
- writeString procedure

Session objects

The Session objects are:

- jmssession.p on page 132
- ptpsession.p on page 133
- pubsubsession.p on page 133

For information on the methods available for the Session objects, see Methods in the Session objects on page 134.

jmssession.p

Starts a JMS session using the unified domain to access both queues and topics in one JMS session object.

Syntax

```
RUN  jms/jmssession.p PERSISTENT SET jmssession (adapterConnection).
```

Parameter

adapterConnection

Desired adapter connection options. For more information, see the Connection options.

Applies to

Session objects

See also

jmssession.p on page 132, ptpsession.p on page 133, pubsubsession.p on page 133

For more information, see Methods in the Session objects on page 134.

ptpsession.p

Starts a JMS session using queues.

Syntax

```
RUN  jms/ptpsession.p PERSISTENT SET ptpsession (adapterConnection).
```

Parameter

adapterConnection

Desired adapter connection options. For more information, see the Connection options.

Applies to

Session objects

See also

jmssession.p on page 132, ptpsession.p on page 133, pubsubsession.p on page 133

For more information, see Methods in the Session objects on page 134.

pubsubsession.p

Starts a JMS session using topics.

Syntax

```
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession (adapterConnection).
```

Parameter

adapterConnection

Desired adapter connection options. For more information, see the Connection options.

Applies to

Session objects

See also

jmssession.p on page 132, ptpsession.p on page 133, pubsubsession.p on page 133

For more information, see Methods in the Session objects on page 134.

Methods in the Session objects

The following table lists the methods in the Session objects.

Table 25: Methods in Session objects

beginSession procedure on page 172	browseQueue procedure on page 172
cancelDurableSubscription procedure on page 173	commitReceive procedure on page 175
commitSend procedure on page 175	createBytesMessage procedure on page 176
createChangeStateConsumer procedure on page 143	createDataSetMessage procedure on page 176
createHeaderMessage procedure on page 177	createMapMessage procedure on page 177
createMessageConsumer procedure on page 178	createMultipartMessage procedure on page 144
createRejectedMessageConsumer procedure on page 144	createStreamMessage procedure on page 179
createTemporaryQueue procedure on page 179	createTemporaryTopic procedure on page 180
createTempTableMessage procedure on page 180	createTextMessage procedure on page 181
createXMLMessage procedure on page 181	deleteSession procedure on page 183
deleteTemporaryQueue procedure on page 184	deleteTemporaryTopic procedure on page 184
getAdapterService function on page 186	getBrokerURL function on page 187
getClientID function on page 190	getClientPersistence function on page 146
getClientTransactionBufferSize function on page 147	getConnectID function on page 190
getConnectionID function on page 191	getConnectionMetaData function on page 191
getConnectionURLs function on page 147	getDefaultPersistency function on page 197
getDefaultPriority function on page 197	getDefaultTimeToLive function on page 198
getFaultTolerant function on page 148	getFaultTolerantReconnectTimeout function on page 149

getFlowToDisk function on page 149	getInitialConnectionTimeout function on page 150
getJMSServerName function on page 206	getLoadBalancing function on page 207
getLocalStoreDirectory function on page 151	getLocalStoreSize function on page 151
getLocalStoreWaitTime function on page 152	For more information, see MultipartMessage and the discussion of the getPassword function in the MultiPartMessage example.
getReconnectInterval function on page 152	getReconnectTimeout function on page 153
getSelectorAtBroker function on page 154	getSequential function on page 154
getShutdownWaitFor function on page 222	getSingleMessageAcknowledgement function on page 155
getTransactedReceive function on page 225	getTransactedSend function on page 225
getUser function on page 226	isFaultTolerant function on page 155
publish procedure on page 232	receiveFromQueue procedure on page 240
recover procedure on page 241	requestReply procedure on page 241
rollbackReceive procedure on page 243	rollbackSend procedure on page 243
sendToQueue procedure on page 244	setAdapterService procedure on page 245
setBrokerURL procedure on page 247	setClientID procedure on page 250
setClientPersistence procedure on page 156	setClientTransactionBufferSize procedure on page 157
setConnectID procedure on page 157	setConnectionFile procedure on page 158
setConnectionURLs procedure on page 158	setDefaultPersistency procedure on page 256
setDefaultPriority procedure on page 256	setDefaultTimeToLive procedure on page 257
setFaultTolerant procedure on page 159	setFaultTolerantReconnectTimeout procedure on page 160
setInitialConnectionTimeout procedure on page 161	setErrorHandler procedure on page 258
setFlowToDisk procedure on page 161	setJMSServerName procedure on page 264
setLoadBalancing procedure on page 265	setLocalStoreDirectory procedure on page 162
setLocalStoreSize procedure on page 163	setLocalStoreWaitTime procedure on page 164
setNoErrorDisplay procedure on page 269	setReconnectInterval procedure on page 167
setReconnectTimeout procedure on page 167	setPassword procedure on page 270

setPingInterval procedure on page 164	setPrefetchCount procedure on page 165
setPrefetchThreshold procedure on page 166	setSelectorAtBroker procedure on page 168
setSequential procedure on page 168	setShutdownWaitFor procedure on page 276
setSingleMessageAcknowledgement procedure on page 169	setTransactedReceive procedure on page 279
setTransactedSend procedure on page 279	setUser procedure on page 280
startReceiveMessages procedure on page 281	stopReceiveMessages procedure on page 282
subscribe procedure on page 282	waitForMessages procedure on page 283

Methods in the Message Consumer objects

The following table lists the methods in the Message Consumer objects.

Table 26: Methods in Message Consumer Objects

acknowledgeAndForward procedure on page 141	deleteConsumer procedure on page 182
getApplicationContext function on page 186	getDestinationName function on page 198
getNoAcknowledge function on page 214	getProcHandle function on page 216
getProcName function on page 216	getReplyAutoDelete function on page 217
getReplyPersistency function on page 218	getReplyPriority function on page 218
getReplyTimeToLive function on page 219	getReuseMessage function on page 220
getSession function on page 221	inErrorHandling function on page 227
inMessageHandling function on page 228	inQueueBrowsing function on page 228
inReplyHandling function on page 229	setApplicationContext procedure on page 245
setNoAcknowledge procedure on page 269	setReplyPersistency procedure on page 271
setReplyPriority procedure on page 272	setReplyTimeToLive procedure on page 272
setReuseMessage procedure on page 274	

For more information, see the Message Consumer objects.

Methods in the Message objects

The following table lists the methods in the Message objects.

Note: Not all methods are in all message types.

Table 27: Methods in Message Objects

addBytesPart procedure on page 170	addMessagePart procedure on page 142
addTextPart procedure on page 170	appendText procedure on page 171
clearBody procedure on page 174	clearProperties procedure on page 174
deleteMessage procedure on page 182	deleteSaxWriter procedure on page 183
endOfStream function on page 185	getBytesCount function on page 187
getBytesPartByID function on page 145	getBytesPartByIndex function on page 145
getBytesToRaw function on page 188	getChar function on page 188
getCharCount function on page 189	getCharProperty function on page 189
getConnectionURLs function on page 147	getContentType on page 148
getDataSet function on page 192	getDate function on page 192
getDateProperty function on page 193	getDateTime function on page 193
getDateTimeProperty function on page 194	getDateTime-TZ function on page 195
getDateTimeTzProperty function on page 195	getDecimal function on page 196
getDecimalProperty function on page 196	getInt function on page 199
getIntProperty function on page 199	getInt64 function on page 200
getInt64Property function on page 200	getItemType function on page 201
getJMSCorrelationID function on page 201	getJMSCorrelationIDAsBytes function on page 202
getJMSDeliveryMode function on page 202	getJMSDestination function on page 203
getJMSExpiration function on page 203	getJMSMessageID function on page 204
getJMSPriority function on page 204	getJMSRedelivered function on page 205
getJMSReplyTo function on page 205	getJMSTimestamp function on page 206
getJMSType function on page 207	getLoadBalancing function on page 207

getLogical function on page 208	getLogicalProperty function on page 208
getLongString function on page 209	getLongStringCP function on page 209
getLongText function on page 210	getLongTextCP function on page 210
getMapNames function on page 211	getMemptr function on page 212
getMessagePartByID function on page 212	getMessagePartByIndex function on page 213
getMessageType function on page 214	getPartCount function on page 215
getPropertyNames function on page 216	getPropertyType function on page 217
getReplyToDestinationType function on page 219	getSaxWriter function on page 221
getSequential function on page 154	getTempTable function on page 222
getText function on page 223	getTextPartByID function on page 223
getTextPartByIndex function on page 224	getTextSegment function on page 224
getX-Document function on page 226	hasReplyTo function on page 227
isMessagePart function on page 229	messageHandler procedure on page 230
moveToNext procedure on page 231	readBytesToRaw procedure on page 233
readChar function on page 234	readDate function on page 234
readDateTime function on page 235	readDateTime-TZ function on page 236
readDecimal function on page 236	readInt function on page 237
readInt64 function on page 238	readLogical function on page 238
readLongString function on page 239	readLongStringCP function on page 239
reset procedure on page 242	setBoolean procedure on page 246
setBooleanProperty procedure on page 247	setByte procedure on page 248
setByteProperty procedure on page 248	setBytesFromRaw procedure on page 249
setChar procedure on page 250	setDataSet procedure on page 251
setDate procedure on page 251	setDateProperty procedure on page 252
setDateTime procedure on page 253	setDateTimeProperty procedure on page 254
setDateTime-TZ procedure on page 254	setDateTimeTzProperty procedure on page 255
setDouble procedure on page 257	setDoubleProperty procedure on page 258

setFloat procedure on page 259	setFloatProperty procedure on page 260
setInt procedure on page 260	setInt64 procedure on page 261
setInt64Property procedure on page 262	setIntProperty procedure on page 261
setJMSCorrelationID procedure on page 262	setJMSCorrelationIDAsBytes procedure on page 263
setJMSReplyTo procedure on page 263	setJMSType procedure on page 265
setLong procedure on page 266	setLongProperty procedure on page 266
setLongString procedure on page 267	setLongText procedure on page 267
setMemptr procedure on page 268	setNoErrorDisplay procedure on page 269
setReplyToDestinationType procedure on page 273	setSaxReader procedure on page 274
setSequential procedure on page 168	setShort procedure on page 275
setShortProperty procedure on page 275	setSingleMessageAcknowledgement procedure on page 169
setString procedure on page 277	setStringProperty procedure on page 277
setTempTable procedure on page 278	setText procedure on page 278
setX-Document procedure on page 280	writeBoolean procedure on page 284
writeByte procedure on page 285	writeBytesFromRaw procedure on page 286
writeChar procedure on page 286	writeDate procedure on page 287
writeDateTime procedure on page 288	writeDateTime-TZ procedure on page 289
writeDouble procedure on page 289	writeFloat procedure on page 290
writeInt procedure on page 290	writeInt64 procedure on page 291
writeLong procedure on page 292	writeLongString procedure on page 292
writeShort procedure on page 293	writeString procedure on page 294

For more information, see the Message objects.

SonicMQ proprietary APIs

The classic ABL API specifically supports all the features supported by the SonicMQ JMS provider. However, the other JMS providers do not support all the capabilities and methods in this API. Here is a list of capabilities that are not supported through the classic ABL API for other third-party JMS providers:

- FaultTolerant
- Client Persistency
- Load Balancing
- Multi-part messages
- Message selector
- Acknowledge And Forward
- Failover
- FlowToDisk

However, these capabilities can be configured through JMS Administered Objects using JNDI. For more information, see Build the AdminObjectFinder for use with JNDI on page 42 and the Configure JMS-provider proprietary features on page 103 chapter.

The following SonicMQ proprietary APIs were supported in the classic ABL API in the OpenEdge SonicMQ Adapter. You can still use their functionality by configuring SonicMQ as your JMS provider. For more information, see Configure your JMS provider on page 39. The Progress OpenEdge JMS Adapter does not support the following methods in Session objects for third-party JMS providers other than SonicMQ:

- `createChangeStateConsumer procedure`
- `createMultipartMessage procedure`
- `createRejectedMessageConsumer procedure`
- `getClientPersistence function`
- `getClientTransactionBufferSize function`
- `getConnectionURLs function`
- `getFaultTolerant function`
- `getFaultTolerantReconnectTimeout function`
- `getFlowToDisk function`
- `getInitialConnectionTimeout function`
- `getLocalStoreDirectory function`
- `getLocalStoreSize function`
- `getLocalStoreWaitTime function`
- `getReconnectInterval function`
- `getReconnectTimeout function`
- `getSelectorAtBroker function`
- `getSequential function`
- `getSingleMessageAcknowledgement function`
- `isFaultTolerant function`
- `setClientPersistence procedure`

- setConnectID procedure
- setConnectionFile procedure
- setConnectionURLs procedure
- setFaultTolerant procedure
- setFaultTolerantReconnectTimeout procedure
- setInitialConnectionTimeout procedure
- setFlowToDisk procedure
- setLocalStoreDirectory procedure
- setLocalStoreSize procedure
- setLocalStoreWaitTime procedure
- setReconnectInterval procedure
- setReconnectTimeout procedure
- setPingInterval procedure
- setPrefetchCount procedure
- setPrefetchThreshold procedure
- setSelectorAtBroker procedure
- setSequential procedure
- setSingleMessageAcknowledgement procedure

The Progress OpenEdge JMS adapter does not support the following method in Message Consumer objects for third-party JMS providers other than SonicMQ:

- acknowledgeAndForward procedure

The Progress OpenEdge JMS adapter does not support the following methods in Message objects for third-party JMS providers other than SonicMQ:

- addMessagePart procedure
- getBytesPartByID function
- getBytesPartByIndex function
- getContentType function

acknowledgeAndForward procedure

Forwards and acknowledges a message in a single operation.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE acknowledgeAndForward.
DEFINE INPUT PARAMETER destinationName AS CHARACTER.
DEFINE INPUT PARAMETER messageH        AS HANDLE.
DEFINE INPUT PARAMETER priority        AS INTEGER.
DEFINE INPUT PARAMETER timeToLive      AS DECIMAL.
DEFINE INPUT PARAMETER persistency     AS CHARACTER.
```

Applies to

Message Consumer objects

Notes

- The message cannot be changed when using the acknowledgeAndForward procedure.
- This procedure applies inside a message event handler.
- The session must be set to `SINGLE_MESSAGE_ACKNOWLEDGE`.
- The procedure expects a destination queue name, the original message handle, and optional message-delivery properties. If the message-delivery properties are set to the `Unknown value (?)`, the procedure uses the original values from the message.
- If the procedure is not successful—for example, if the destination does not exist—the message is not acknowledged and eventually returns to the queue.

See also

acknowledgeAndForward procedure on page 141, setSingleMessageAcknowledgement procedure on page 169, getSingleMessageAcknowledgement function on page 155, setNoAcknowledge procedure on page 269 , getNoAcknowledge function on page 214

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.

addMessagePart procedure

Adds a SonicMQ message to a `MultipartMessage`.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE addMessagePart.
DEFINE INPUT PARAMETER messagePartH AS HANDLE.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
```

Applies to

Message objects

Notes

- The Sonic message is created as usual.
- Its content type is defined by Sonic. The content-ID string (represented by *contentIDString*) sets the content ID of the part and is used to identify it.
- To conserve resources, after calling this procedure, the application must delete the message-part handle (represented by *messagePartH*).

See also

createMultipartMessage procedure on page 144, addBytesPart procedure on page 170, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getContentType on page 148, getPartCount function on page 215, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

createChangeStateConsumer procedure

Creates a "Change State" handle to contain the state of the SonicMQ Broker connection changes.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE createChangeStateConsumer
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHARACTER.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Fault Tolerant connections.
- Call createChangeStateConsumer procedure on page 143 after beginSession procedure on page 172.
- In the "Change State" handler, the character header property "state" will contain one of the following values: `"active"`, `"reconnecting"`, `"failed"`, or `"closed"`.
- Setting up a change-state handler is optional. If not done, the default is to not notify the application of state changes.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

createMultipartMessage procedure

Creates a `MultipartMessage`.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE createMultipartMessage.
DEFINE OUTPUT PARAMETER messageH AS HANDLE.
```

Applies to

Session objects

See also

addBytesPart procedure on page 170, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getContentType on page 148, getPartCount function on page 215, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

createRejectedMessageConsumer procedure

Creates a Message Consumer to handle all rejected messages.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE createRejectedMessageConsumer.
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHARACTER.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

Applies to

Session objects

Notes

- Only applicable for Client Persistence.
- Call createRejectedMessageConsumer procedure on page 144 after beginSession procedure on page 172.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

getBytesPartByID function

Retrieves a bytes part and returns the content type as a CHARACTER string.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getBytesPartByID RETURNS CHARACTER
  (INPUT contentID AS INTEGER, OUTPUT memPtr AS MEMPTR).
```

Applies to

Message objects

Notes

- Before calling this function, call SET-SIZE to free any memory allocated by the MEMPTR.
- The bytes part does not undergo any code-page conversion. If it consists of text data, it is encoded in UTF-8. To encode it differently, either convert the code page manually or use one of the getTextPartBy... functions.

See also

createMultipartMessage procedure on page 144, addBytesPart procedure on page 170, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getPartCount function on page 215, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

getBytesPartByIndex function

Retrieves a bytes part and returns the content type as a CHARACTER string.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getMessagePartByIndex RETURNS CHARACTER
  (INPUT iIndex AS INTEGER, OUTPUT memPtr AS MEMPTR).
```

Applies to

Message objects

Notes

- Before calling this function, call SET-SIZE to free any memory allocated by the MEMPTR.
- The bytes part does not undergo any code-page conversion. If it consists of text data, it is encoded in UTF-8. To encode it differently, either convert the code page manually or use one of the getTextPartBy... functions.

See also

createBytesMessage procedure on page 176, setMemptr procedure on page 268, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, getBytesCount function on page 187, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239, getMemptr function on page 212

For more information, see the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

For more information, see the MultipartMessage and the MultiPartMessage example.

getClientPersistence function

Returns the state of client persistence.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getClientPersistence RETURNS LOGICAL.
```

Applies to

Session objects

Notes

- Only applicable for Client Persistence.

- The default is FALSE.

See also

createRejectedMessageConsumer procedure on page 144

, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

getClientTransactionBufferSize function

Returns the client buffer size in bytes for Fault Tolerant transacted messages in memory.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getClientTransactionBufferSize RETURNS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only)

Notes

- Only applicable for Fault Tolerant connections.
- The default is 0. A value of 0 tells the SonicMQ client to use the default value as determined by the SonicMQ Broker. This value is the size of the buffer used by the SonicMQ Broker.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

getConnectionURLs function

Returns a comma-separated list of SonicMQ Broker URLs that the client will try to connect to.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getConnectionURLs RETURNS CHARACTER.
```

Applies to

Session objects

See also

setConnectionURLs procedure on page 158, getConnectionURLs function on page 147, setSequential procedure on page 168, getSequential function on page 154

For more information, see the Managing fail-over support.

getContentType

Retrieves the content type of the message part corresponding to the index in a `MultipartMessage`.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getContentType RETURNS CHARACTER (INPUT iIndex AS INTEGER).
```

Applies to

Message objects

See also

createMultipartMessage procedure on page 144, addBytesPart procedure on page 170, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getPartCount function on page 215, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

getFaultTolerant function

Returns the current Fault Tolerant setting.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getFaultTolerant RETURNS LOGICAL.
```

Applies to

Session objects (ClientConnect and ServerConnect only)

Notes

- Only applicable for Fault Tolerant connections.
- The default is FALSE.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

getFaultTolerantReconnectTimeout function

Returns the Fault Tolerant reconnection timeout.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getFaultTolerantReconnectTimeout RETURNS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only)

Notes

- Only applicable for Fault Tolerant connections.
- The default is 60 seconds.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

getFlowToDisk function

This function returns the current flow-to-disk setting.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getFlowToDisk RETURNS INTEGER.
```

Applies to

Session objects

See also

setFlowToDisk procedure on page 161

For more information on this SonicMQ feature, see the "Flow to Disk" section in the "SonicMQ Client Sessions" chapter of the *SonicMQ Application Programming Guide*.

getInitialConnectionTimeout function

Returns the initial Fault Tolerant reconnection timeout in seconds.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getInitalConnectionTimeout RETURNS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only)

Notes

- Only applicable for Fault Tolerant connections.
- The default is 30 seconds.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

getLocalStoreDirectory function

Returns the directory that will be used by the adapter to persist messages.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getLocalStoreDirectory RETURNS CHARACTER.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- This value is set by a call to setLocalStoreDirectory procedure on page 162.
- If setLocalStoreDirectory procedure on page 162 was never called, the value is the `Unknown value (?)`.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

getLocalStoreSize function

Returns the maximum size of the local store in kilobytes.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getLocalStoreSize RETURNS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- The default size is `10000 (10MB)`.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

getLocalStoreWaitTime function

Returns the wait interval before Client Persistence begins.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getLocalStoreWait RETURNS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- The default is 5 seconds.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

getReconnectInterval function

Returns the interval for reconnection attempts in seconds.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getReconnectInterval RETURNS INTEGER
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- The default is 30 seconds.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

getReconnectTimeout function

Returns the timeout for reconnection attempts in minutes.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getReconnectTimeout RETURNS INTEGER
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- The default is 0 indicating no timeout.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

getSelectorAtBroker function

Gets the broker selector setting.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getSelectorAtBroker RETURNS LOGICAL.
```

Applies to

Session objects

Notes

The default is FALSE.

See also

setSelectorAtBroker procedure on page 168, getSelectorAtBroker function on page 154

For more information, see the Message selectors.

getSequential function

Returns a LOGICAL value indicating how a fail-over list is used.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getSequential RETURNS LOGICAL.
```

Applies to

Session objects

Notes

- When using a fail-over list, clients try to connect to brokers in the list either sequentially or randomly.
- getSequential function on page 154 returns TRUE if connection attempts are sequential and FALSE if connection attempts are random.

See also

setConnectionURLs procedure on page 158, getConnectionURLs function on page 147, setSequential procedure on page 168, getSequential function on page 154

For more information, see the Managing fail-over support.

getSingleMessageAcknowledgement function

Returns a logical value indicating whether a client session is configured to use single-message acknowledgement.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
FUNCTION getSingleMessageAcknowledegment RETURNS LOGICAL.
```

Applies to

Session objects

Notes

- getSingleMessageAcknowledgement function on page 155 returns TRUE if the client session is configured to use single-message acknowledgement.
- getSingleMessageAcknowledgement function on page 155 returns FALSE if the client session is not so configured.

See also

acknowledgeAndForward procedure on page 141, setSingleMessageAcknowledgement procedure on page 169, getSingleMessageAcknowledgement function on page 155, setNoAcknowledge procedure on page 269, getNoAcknowledge function on page 214

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.

isFaultTolerant function

Determines if the SonicMQ Broker connected supports Fault Tolerance.

Syntax

```
FUNCTION isFaultTolerant RETURNS LOGICAL
```

Applies to

Session objects

Notes

- Only applicable for Fault Tolerant connections.
- This function must be called after beginSession procedure on page 172 is called.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

Note: The JMS providers other than SonicMQ do not support this method.

setClientPersistence procedure

Sets client persistence.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setClientPersistence.
DEFINE INPUT PARAMETER enabled AS LOGICAL.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- You must call setClientID procedure on page 250 to use client persistence.
- If the default is FALSE, client persistence is not enabled.
- You cannot change client persistence for an active session.
- Call setClientPersistence procedure on page 156 prior to calling beginSession procedure on page 172.

- To change the value of client persistence, you must stop and restart the session for any changes to take effect.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

setClientTransactionBufferSize procedure

Sets the SonicMQ client buffer size in bytes for Fault Tolerant transacted messages in memory.

Syntax

```
PROCEDURE setClientTransactionBuffersize.
DEFINE INPUT PARAMETER sz AS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Fault Tolerant connections.
- Call setClientTransactionBufferSize procedure on page 157 before beginSession procedure on page 172 is called.
- A value of 0 tells the SonicMQ client to use the default value as determined by the SonicMQ Broker. This value is the size of the buffer used by the SonicMQ Broker.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

setConnectID procedure

Sets the connection ID between the Sonic client and Sonic broker.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setConnectID.
DEFINE INPUT PARAMETER connectID AS CHARACTER.
```

Applies to

Session objects.

Notes

Sets the connection ID between the Sonic client and Sonic broker. If not explicitly set, Sonic uses an internal value.

See also

getConnectID function on page 190

setConnectionFile procedure

Sets the connection file.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setConnectionFile.
DEFINE INPUT PARAMETER filename AS CHARACTER.
```

Applies to

Session objects.

Notes

- This file contains all the serialized connection object information used to connect to a Sonic MQ Broker.
- Parameter values in the connection file are superseded by any parameter set through a call to the corresponding `set<parameter>` procedure.
- Connection parameter values are fixed when beginSession procedure on page 172 is called.

See also

For more information, see the Using serialized connection objects.

setConnectionURLs procedure

Specifies a list of broker URLs for the client to try to connect to.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setConnectionURLs.
DEFINE INPUT PARAMETER brokerList AS CHARACTER.
```

Parameter

brokerList

A comma-separated list of Sonic Broker URLs for the client to use for connecting.

Applies to

Session objects

Notes

- If brokerList is not set to the `Unknown value (?)`, it overrides the URL specified by setBrokerURL procedure on page 247.
- Call this procedure instead of setBrokerURL procedure on page 247 when there is a list of broker URLs.

See also

setConnectionURLs procedure on page 158, getConnectionURLs function on page 147, setSequential procedure on page 168, getSequential function on page 154

For more information, see the Managing fail-over support.

setFaultTolerant procedure

Enables or disables Fault Tolerance for the session.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setFaultTolerant.
DEFINE INPUT PARAMETER enable AS LOGICAL.
```

Applies to

Session objects (ClientConnect and ServerConnect only)

Notes

- Only applicable for Fault Tolerant connections.

- A value of TRUE will enable fault-tolerance and a value of FALSE will disable it. If default is FALSE, fault-tolerance is not enabled.
- This procedure must be called before beginSession procedure on page 172 is called.
- You cannot change fault-tolerance for an active session. You must stop and restart the session for any changes to have an effect.
- The application must also call setConnectionURLs procedure on page 158 to provide a list of broker URLs to be connected to when the current connection fails. SonicMQ will connect to the URLs in the order they are listed, starting at the beginning of the list.
- setSequential procedure on page 168 may be called to connect to the urls in the order they are listed starting at a random place in the list. See the SonicMQ documentation for details.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

setFaultTolerantReconnectTimeout procedure

Sets a time limit in seconds on reconnection attempts.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setFaultTolerantReconnectTimeout.
DEFINE INPUT PARAMETER seconds AS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Fault Tolerant connections.
- The default is 60 seconds. A timeout value of 0 indicates no timeout (that is, the reconnect will be attempted indefinitely).
- This procedure must be called before beginSession procedure on page 172 is called.
- The URLs specified in setConnectionURLs procedure on page 158 are used to attempt reconnection to a SonicMQ Broker.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

setFlowToDisk procedure

Enables or disables the SonicMQ flow-to-disk functionality.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setFlowToDisk.
   DEFINE INPUT PARAMETER val AS INTEGER NO-UNDO.
```

Applies to

Session objects

Notes

- When enabled, the SonicMQ broker saves messages to disk if the client is blocked and cannot receive the messages.
- You can set up the SonicMQ broker to have this feature on or off by default.
- The input parameter can have the following possible values:
  - **0** specifies to use the broker setting.
  - **1** turns on flow-to-disk.
  - **2** turns off flow-to-disk.
- This procedure can only be called before the session is started with beginSession procedure on page 172.
- The SonicMQ broker only uses the flow-to-disk feature for pub/sub messages.

See also

getFlowToDisk function on page 149

For more information on this SonicMQ feature, see the "Flow to Disk" section in the "SonicMQ Client Sessions" chapter of the *SonicMQ Application Programming Guide*.

setInitialConnectionTimeout procedure

Sets a time limit in seconds on the initial Fault tolerant connection attempt to the SonicMQ Broker.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setInitialConnectionTimeout.
DEFINE INPUT PARAMETER seconds AS INTEGER.
```

Applies to

Message objects

Notes

- Only for Fault Tolerant connections.
- The default is 30 seconds. The timeout value specifies the timeout in seconds. A timeout value of 0 indicates no timeout (that is, the connect will be attempted indefinitely).
- This procedure must be called before beginSession procedure on page 172 is called.
- The URLs specified in setConnectionURLs procedure on page 158 are used to attempt to connect to a SonicMQ Broker.
- If the time limit is reached without being able to connect, the application calls the Adapter error handler.

See also

setFaultTolerant procedure on page 159, getFaultTolerant function on page 148, isFaultTolerant function on page 155, createChangeStateConsumer procedure on page 143, setFaultTolerantReconnectTimeout procedure on page 160, getFaultTolerantReconnectTimeout function on page 149, setInitialConnectionTimeout procedure on page 161, getInitialConnectionTimeout function on page 150, setClientTransactionBufferSize procedure on page 157, getClientTransactionBufferSize function on page 147

For more information, see the Fault tolerance and the Fault tolerance.

setLocalStoreDirectory procedure

Sets the directory that will be used by the adapter to persist messages.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setLocalStoreDirectory
DEFINE INPUT PARAMETER localStoreDir AS CHARACTER.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- The default is the current working directory.
- The directory used will be appended with the client ID that is set to ensure uniqueness when operating in MQ-ServerConnect mode.
- Errors for this call will occur after the call to beginSession procedure on page 172.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

setLocalStoreSize procedure

Sets the maximum size of the local store in kilobytes.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setLocalStoreSize
DEFINE INPUT PARAMETER storesize AS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- The default size is `1000 (10MB)`.
- Errors for this call will occur after the call to beginSession procedure on page 172.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

setLocalStoreWaitTime procedure

Sets the interval in seconds before Client Persistence begins.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setLocalStoreWait.
DEFINE INPUT PARAMETER interval AS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only).

Notes

- Only applicable for Client Persistence.
- The default is 5, which means the adapter will wait 5 seconds then persist messages to disk.
- Errors for this call will occur after the call to beginSession procedure on page 172.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

setPingInterval procedure

Specifies the interval in seconds for the JMS Adapter to actively ping the SonicMQ Broker so communication failure can be detected promptly.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setPingInterval.
DEFINE INPUT PARAMETER pingInterval AS INTEGER.
```

Parameter

pingInterval

The interval (in seconds).

Applies to

Session objects

Notes

- No pinging is performed by default.
- The `setPingInterval` functionality is a SonicMQ extension (see *SonicMQ Programming Guide*).
- A *pingInterval* value can also be specified in the `ubroker.properties` file for all clients by using the `srvrStartupParam` property of the OpenEdge Adapter for SonicMQ, as shown:

```
srvrStartupParam=pingInterval=3
```

- If you are connecting using ClientConnect mode, the *pingInterval* value must be set in the `JavaTools.properties` file under the `[Adapter.CC.cc1]` section, for example:

```
[Adapter.CC.cc1]
...
srvrStartupParam=pingInterval=3
```

- The `setPingInterval` procedure must be called before beginSession procedure on page 172 is called.

See also

For more information, see the Setting and getting JMS connection and session attributes, and also see *SonicMQ Programming Guide*.

setPrefetchCount procedure

Sets the number of messages a SonicMQ client can retrieve in a single operation from a queue containing multiple messages.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setPrefetchCount.
DEFINE INPUT PARAMETER count AS INTEGER.
```

Applies to

Session objects

Notes

- The default is 3. For example, a *count* of 3 means that the Sonic client retrieves up to three message from a queue.
- If this procedure is called after beginSession procedure on page 172 is called, an error is raised.

See also

setPrefetchCount procedure on page 165, setPrefetchThreshold procedure on page 166

For more information, see the Controlling flow of messages.

setPrefetchThreshold procedure

Determines when the SonicMQ client goes back to the broker for more messages.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setPrefetchThreshold.
DEFINE INPUT PARAMETER threshold AS INTEGER.
```

Parameter

threshold

Prefetch threshold.

Applies to

Session objects for `ptpsession`

Notes

- The default is 1. For example, a *threshold* value of 1 means that Sonic does not go back to the broker for more messages until the last message has been delivered.
- If this procedure is called after beginSession procedure on page 172 is called, an error is raised.

See also

setPrefetchCount procedure on page 165, setPrefetchThreshold procedure on page 166

For more information, see the Controlling flow of messages.

setReconnectInterval procedure

Sets the interval in seconds between reconnect attempts.

Syntax

```
PROCEDURE setReconnectInterval.
DEFINE INPUT PARAMETER interval AS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only)

Note: The JMS providers other than SonicMQ do not support this method.

Notes

- Only applicable for Client Persistence.
- The default is 30, which means the adapter will attempt to reconnect to an MQ Broker every 30 seconds.
- Errors for this call will occur after the call to beginSession procedure on page 172.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

setReconnectTimeout procedure

Sets the maximum amount of time in minutes that the client will attempt to reconnect to a broker.

Note: The JMS providers other than SonicMQ do not support this method.

Syntax

```
PROCEDURE setReconnectTimeout.
DEFINE INPUT PARAMETER timout AS INTEGER.
```

Applies to

Session objects (ClientConnect and ServerConnect only)

Notes

- Only applicable for Client Persistence.
- The default is 0 meaning there is no timeout.
- If the value is set, an asynchronous error will be sent to the OpenEdge application after the timeout has expired and no further reconnects will be attempted.
- Errors for this call will occur after the call to beginSession procedure on page 172.

See also

createRejectedMessageConsumer procedure on page 144, setClientPersistence procedure on page 156, getClientPersistence function on page 146, setLocalStoreDirectory procedure on page 162, getLocalStoreDirectory function on page 151, setLocalStoreSize procedure on page 163, getLocalStoreSize function on page 151, setLocalStoreWaitTime procedure on page 164, getLocalStoreWaitTime function on page 152, setReconnectTimeout procedure on page 167, getReconnectTimeout function on page 153, setReconnectInterval procedure on page 167, getReconnectInterval function on page 152

For more information, see the Client persistence and the Client persistence.

setSelectorAtBroker procedure

Sets message filtering at the SonicMQ Broker (instead of the SonicMQ client).

Syntax

```
PROCEDURE setSelectorAtBroker.
DEFINE INPUT PARAMETER seq AS LOGICAL.
```

Applies to

Session objects

Notes

- Since the SonicMQ Broker typically runs on a machine with more resources than the machine running the SonicMQ client, it is desirable to have the SonicMQ Broker provide this filtering instead of the SonicMQ client.
- SonicMQ messages can be filtered so that only those meeting a specific criteria will be received. For point-to-point sessions, this filtering is always done by the SonicMQ Broker. For publish/subscribe sessions, this filtering is done by the SonicMQ client by default.
- This procedure must be called before beginSession procedure on page 172.

See also

setSelectorAtBroker procedure on page 168, getSelectorAtBroker function on page 154

For more information, see the Message selectors.

setSequential procedure

Sets the method the client application will use to connect to the broker.

Syntax

```
PROCEDURE setSequential.
DEFINE INPUT PARAMETER seq AS LOGICAL.
```

Applies to

Session objects

Notes

- Sonic lets clients try to connect to brokers in a connection list in two ways:
  - **Sequentially** — Starting with the first broker in the list and working sequentially
  - **Randomly** — Repeatedly picking a broker randomly
- The default is TRUE, which tells clients to try to connect sequentially.
- To attempt load balancing, set seq to FALSE, which tells clients to try to connect randomly.

See also

setConnectionURLs procedure on page 158, getConnectionURLs function on page 147, setSequential procedure on page 168, getSequential function on page 154

For more information, see the Managing fail-over support.

setSingleMessageAcknowledgement procedure

Turns on single-message acknowledgement for a client session.

Syntax

```
PROCEDURE setSingleMessageAcknowledgement.
DEFINE INPUT PARAMETER ackMethod as LOGICAL.
```

Applies to

Session objects

Notes

- If a session is configured to use single-message acknowledgement, the following rules apply:
  - Groups of messages cannot be acknowledged in one operation.
  - Acknowledge-and-forward can be used.
- This procedure must be called before beginSession procedure on page 172. Otherwise, an error is raised.

See also

acknowledgeAndForward procedure on page 141, setSingleMessageAcknowledgement procedure on page 169, getSingleMessageAcknowledgement function on page 155, setNoAcknowledge procedure on page 269, getNoAcknowledge function on page 214

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.

addBytesPart procedure

Adds any arbitrary part to a `MultipartMessage`.

Syntax

```
PROCEDURE addBytesPart.
DEFINE INPUT PARAMETER memptr AS MEMPTR.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
```

Applies to

Message objects

Notes

- The part can be text or binary. The Sonic message is created as usual.
- A content type and a content ID must be specified.
- To conserve resources, after calling this procedure, the application must delete the memory pointer (represented by *memptr*).

See also

createMultipartMessage procedure on page 144, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getPartCount function on page 215, getContentType on page 148, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

addTextPart procedure

Adds a text part to a `MultipartMessage`.

Syntax

```
PROCEDURE addTextPart.
DEFINE INPUT PARAMETER charString        AS CHARACTER.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString   AS CHARACTER.
```

Applies to

Message objects

Notes

The method resembles addBytesPart procedure on page 170 except that it takes a CHARACTER string instead of a MEMPTR.

See also

createMultipartMessage procedure on page 144, addBytesPart procedure on page 170, addMessagePart procedure on page 142, isMessagePart function on page 229, getContentType on page 148, getPartCount function on page 215, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

appendText procedure

Appends text to the message in write-only mode using several calls to overcome the OpenEdge 32K limit on the number of characters.

Syntax

```
PROCEDURE appendText.
DEFINE INPUT PARAMETER textValue AS CHARACTER.
```

Applies to

Message objects

Notes

- This procedure is designed for use in conjunction with setText procedure on page 278.
- This procedure is not needed when using setLongText procedure on page 267, which writes LONGCHAR data.

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

beginSession procedure

Connects to the Progress OpenEdge JMS Adapter and starts a JMS connection and session.

Syntax

```
PROCEDURE beginSession.
```

Applies to

Session objects

Notes

- If beginSession procedure on page 172 returns an error, the Session object is automatically deleted.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

beginSession procedure on page 172, getSession function on page 221, deleteSession procedure on page 183

For more information, see Connect to the Progress OpenEdge JMS Adapter and Establish session control.

browseQueue procedure

Allows applications to view messages in a queue without consuming them.

Syntax

```
PROCEDURE browseQueue.
DEFINE INPUT PARAMETER queueName AS CHARACTER.
DEFINE INPUT PARAMETER messageSelector AS CHARACTER.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

Parameters

queueName

The queue from which the messages are received.

messageSelector

A message selector.

messageConsumer

A Message Consumer object, which handles the messages asynchronously.

Applies to

Session objects

Notes

- This procedure receives (for browsing) all messages currently in the queue in the *messageConsumer* object.
- Browsed messages are not removed from the queue or acknowledged and are not subject to the transactional context of the session. (For more information, see the *Java Message Service* specification and the *SonicMQ Programming Guide* on queue browsing.)
- The session need not run startReceiveMessages procedure on page 281 to browse messages on a queue.
- This procedure executes remotely (sends a message to the OpenEdge Adapter for SonicMQ).

See also

browseQueue procedure on page 172, receiveFromQueue procedure on page 240 , sendToQueue procedure on page 244

For more information, see theSending messages to a queue, the Receiving messages from a queue, and the Methods unique to Point-to-Point messaging.

For an example, see the PTP message example.

cancelDurableSubscription procedure

Cancels a durable subscription.

Syntax

```
PROCEDURE cancelDurableSubscription.
DEFINE INPUT PARAMETER subscriptionName AS CHARACTER.
```

Parameter

subscriptionName

Name of durable subscription.

Applies to

Session objects

Notes

- It is an error to call this procedure if there is an active Message Consumer for the subscription. Call deleteConsumer procedure on page 182 first to delete the Message Consumer.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

cancelDurableSubscription procedure on page 173 , publish procedure on page 232, subscribe procedure on page 282

For more information, see the Publishing messages to a topic, the Subscribing to a topic, the Durable subscriptions, and the Methods unique to Pub/Sub messaging.

For an example, see the Pub/Sub messaging example on page 85.

clearBody procedure

Clears the body of a message, keeping header and property values unchanged, and changes the mode of a message from read-only to write-only mode.

Syntax

```
PROCEDURE clearBody.
```

Applies to

Message objects

Notes

clearBody procedure on page 174 transfers a `StreamMessage`, `TextMessage`, `XMLMessage`, `BytesMessage`, `TempTableMessage`, or `DataSetMessage` to write-only mode.

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

clearProperties procedure

Clears the properties of the message, keeping the header and body values unchanged.

Syntax

```
PROCEDURE clearProperties.
```

Applies to

Message objects

See also

clearBody procedure on page 174

For more information, see the Accessing message properties.

commitReceive procedure

Acknowledges all messages received up to that point in the current transaction.

Syntax

```
PROCEDURE commitReceive.
```

Applies to

Session objects

Notes

- It is an error to call this method in a Session object that is not transacted for receiving.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

commitSend procedure on page 175, commitReceive procedure on page 175. rollbackSend procedure on page 243, rollbackReceive procedure on page 243 , recover procedure on page 241

For more information, see the Transaction and recovery procedures.

commitSend procedure

Sends all messages published (or sent to a queue) up to that point in the current transaction.

Syntax

```
PROCEDURE commitSend.
```

Applies to

Session objects

Notes

- It is an error to call this method in a Session object that is not transacted for sending.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

commitSend procedure on page 175, commitReceive procedure on page 175. rollbackSend procedure on page 243, rollbackReceive procedure on page 243, recover procedure on page 241

For more information, see the Transaction and recovery procedures.

createBytesMessage procedure

Creates a new `BytesMessage`.

Syntax

```
PROCEDURE createBytesMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createBytesMessage procedure on page 176 , setMemptr procedure on page 268, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, getBytesCount function on page 187, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239, getMemptr function on page 212

For more information, see the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

createDataSetMessage procedure

Creates a new `DataSetMessage`.

Syntax

```
PROCEDURE createDataSetMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createDataSetMessage procedure on page 176, setDataSet procedure on page 251, getDataSet function on page 192

For an example, see the DataSetMessage.

createHeaderMessage procedure

Creates a new `HeaderMessage`.

Syntax

```
PROCEDURE createHeaderMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

For more information, see the HeaderMessage.

createMapMessage procedure

Creates a new `MapMessage`.

Syntax

```
PROCEDURE createMapMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

createMessageConsumer procedure

Creates a new Message Consumer object.

Syntax

```
PROCEDURE createMessageConsumer.
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHARACTER.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

Parameters

procHandle

The handle to a procedure *procName*.

procName

The name of an internal procedure for handling messages.

consumerHandle

The new Message Consumer object.

Applies to

Session objects

Notes

The application must pass the name of an internal procedure for handling messages and the handle to the internal procedure to createMessageConsumer procedure on page 178.

See also

createMessageConsumer procedure on page 178, deleteConsumer procedure on page 182, messageHandler procedure on page 230, waitForMessages procedure on page 283

For more information see the Message Consumer objects, the Consuming messages, the Terminating the Message Consumer object, and the Creating a message handler process.

createStreamMessage procedure

Creates a new `StreamMessage`.

Syntax

```
PROCEDURE createStreamMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

createTemporaryQueue procedure

Creates a temporary queue on the JMS-provider Broker and returns the queue name.

Syntax

```
PROCEDURE createTemporaryQueue.
DEFINE OUTPUT PARAMETER qname AS CHARACTER.
```

Applies to

Session objects

Notes

- An error is returned if beginSession procedure on page 172 has not yet been called.

- An error is returned if the temporary queue cannot be created.

See also

createTemporaryQueue procedure on page 179, deleteTemporaryQueue procedure on page 184, createTemporaryTopic procedure on page 180, deleteTemporaryTopic procedure on page 184

For more information, see the Temporary queues and the Temporary topic.

createTemporaryTopic procedure

Creates a temporary topic on the JMS-provider Broker and returns the topic name.

Syntax

```
PROCEDURE createTemporaryTopic.
DEFINE OUTPUT PRARMETER tname AS CHARACTER.
```

Applies to

Session objects

Notes

- An error is returned if beginSession procedure on page 172 has not yet been called.
- An error is returned if the temporary topic cannot be created.

See also

createTemporaryQueue procedure on page 179, deleteTemporaryQueue procedure on page 184, createTemporaryTopic procedure on page 180, deleteTemporaryTopic procedure on page 184

For more information, see the Temporary queues and the Temporary topic.

createTempTableMessage procedure

Creates a new `TempTableMessage`.

Syntax

```
PROCEDURE createTempTableMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createTempTableMessage procedure on page 180, setTempTable procedure on page 278, getTempTable function on page 222

For more information, see the TempTableMessage.

createTextMessage procedure

Creates a new `TextMessage` object.

Syntax

```
PROCEDURE createTextMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage and the StreamMessage.

createXMLMessage procedure

Creates a new `XMLMessage`.

Syntax

```
PROCEDURE createXMLMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Applies to

Session objects

See also

createXMLMessage procedure on page 181, setX-Document procedure on page 280, getX-Document function on page 226, setSaxReader procedure on page 274, getSaxWriter function on page 221, deleteSaxWriter procedure on page 183

For more information, see the XMLMessage.

deleteConsumer procedure

Deletes the Message Consumer object.

Syntax

```
PROCEDURE deleteConsumer.
```

Applies to

Message Consumer objects

Notes

- In the Pub/Sub domain, deleteConsumer procedure on page 182 cancels the subscription. In the PTP domain, deleteConsumer procedure on page 182 removes the association with a queue.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

createMessageConsumer procedure on page 178, deleteConsumer procedure on page 182, messageHandler procedure on page 230, waitForMessages procedure on page 283

For more information see the Message Consumer objects, the Consuming messages, the Terminating the Message Consumer object, and the Creating a message handler process.

deleteMessage procedure

Deletes a message (including a `TempTableMessage` and `DataSetMessage`) and deallocates all of its memory and resources.

Syntax

```
PROCEDURE deleteMessage.
```

Applies to

Message objects

See also

For more information, see the Deleting messages.

deleteSaxWriter procedure

Terminates the SAX-WRITER and deletes the SAX-WRITER handle created by getSaxWriter function on page 221.

Syntax

```
PROCEDURE deleteSaxWriter.
INPUT PARAMETER hdl AS HANDLE.
```

Applies to

Message objects

Notes

- This procedure calls setLongText procedure on page 267 to place the XML created by the SAX-WRITER calls into the XMLMessage.
- The caller then sends the XMLMessage using the appropriate OpenEdge Adapter for SonicMQ calls.
- An error is returned if the specified handle is not a SAX-WRITER handle, by setLongText procedure on page 267, and by DELETE OBJECT.

See also

createXMLMessage procedure on page 181, setX-Document procedure on page 280, getX-Document function on page 226, setSaxReader procedure on page 274, getSaxWriter function on page 221, deleteSaxWriter procedure on page 183

For more information, see the XMLMessage.

deleteSession procedure

Closes a session and its underlying connection and deletes the session procedure.

Syntax

```
PROCEDURE deleteSession.
```

Applies to

Session objects

Notes

This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

beginSession procedure on page 172, getSession function on page 221, deleteSession procedure on page 183

For more information, see the Establishing session control.

deleteTemporaryQueue procedure

Deletes the temporary queue on the JMS-adapter Broker.

Syntax

```
PROCEDURE deleteTemporaryQueue.
DEFINE INPUT PARAMETER qname AS CHAR.
```

Applies to

Session objects

Notes

- Any Message Consumers for the temporary queue must be deleted with deleteConsumer procedure on page 182 before calling this procedure.
- An error is returned if beginSession procedure on page 172 has not yet been called.
- An error is returned if the temporary queue cannot be deleted.

See also

createTemporaryQueue procedure on page 179, deleteTemporaryQueue procedure on page 184, createTemporaryTopic procedure on page 180, deleteTemporaryTopic procedure on page 184

For more information, see the Temporary queues and the Temporary topic.

deleteTemporaryTopic procedure

Deletes the temporary topic on the JMS-adapter Broker.

Syntax

```
PROCEDURE deleteTemporaryTopic.
DEFINE INPUT PARAMETER tname AS CHARACTER.
```

Applies to

Session objects

Notes

- Any Message Consumers for the temporary topic must be deleted with deleteConsumer procedure on page 182 before calling this procedure.
- An error is returned if beginSession procedure on page 172 has not yet been called.
- An error is returned if the temporary topic cannot be deleted.

See also

createTemporaryQueue procedure on page 179, deleteTemporaryQueue procedure on page 184, createTemporaryTopic procedure on page 180, deleteTemporaryTopic procedure on page 184

For more information, see the Temporary queues and the Temporary topic.

endOfStream function

Returns TRUE if the application retrieved the last text segment, the last item of a stream, or the last byte segment.

Syntax

```
FUNCTION endOfStream RETURNS LOGICAL.
```

Applies to

Message objects

Notes

- An application should not call endOfStream function on page 185 if it used getMemptr function on page 212 for extracting the data.
- endOfStream function on page 185 is also used with the `TextMessage`, `StreamMessage`, and `BytesMessage` message types.

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

createBytesMessage procedure on page 176, setMemptr procedure on page 268, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, getBytesCount function on page 187, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239, getMemptr function on page 212

For more information, see the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

getAdapterService function

Returns the service name under which the Progress OpenEdge JMS Adapter BrokerConnect is registered with.

Syntax

```
FUNCTION getAdapterService RETURNS CHARACTER.
```

Applies to

Session objects (for BrokerConnect only)

Notes

If setAdapterService procedure on page 245 was not called, Null is returned.

See also

setAdapterService procedure on page 245, getAdapterService function on page 186

For more information, see the Setting and getting JMS connection and session attributes.

getApplicationContext function

Returns application context information.

Syntax

```
FUNCTION getApplicationContext RETURNS HANDLE.
```

Applies to

Message Consumer objects

See also

setApplicationContext procedure on page 245, getApplicationContext function on page 186

For more information, see the Accessing message handler information and the Creating a message handler process.

getBrokerURL function

Returns the value set by the preceding setBrokerURL procedure on page 247.

Syntax

```
FUNCTION getBrokerURL RETURNS CHARACTER.
```

Applies to

Session objects

Notes

If setBrokerURL procedure on page 247 was not called, Null is returned.

See also

setBrokerURL procedure on page 247, getBrokerURL function on page 187

For more information, see the Setting and getting JMS connection and session attributes.

getBytesCount function

Returns the number of bytes in a `BytesMessage`.

Syntax

```
FUNCTION getBytesCount RETURNS INTEGER.
```

Applies to

Message objects

See also

createBytesMessage procedure on page 176, setMemptr procedure on page 268, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, getBytesCount function on page 187, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239, getMemptr function on page 212

For more information, see the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

getBytesToRaw function

Gets a bytes item from a `MapMessage`.

Syntax

```
FUNCTION getBytesToRaw RETURNS RAW (itemName AS CHARACTER).
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getChar function

Gets an item of any data type except `byte` from a `MapMessage`.

Syntax

```
FUNCTION getChar RETURNS CHARACTER (itemName AS CHARACTER).
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getCharCount function

Returns the total number of characters in a message.

Syntax

```
FUNCTION getCharCount RETURNS INTEGER.
```

Applies to

Message objects

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

getCharProperty function

Returns message properties of any data type.

Syntax

```
FUNCTION getCharProperty RETURNS CHARACTER (propertyName AS CHARACTER).
```

Applies to

Message objects

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

getClientID function

Returns the client ID value for the JMS-provider Broker connection.

Syntax

```
FUNCTION getClientID RETURNS CHARACTER.
```

Applies to

Session objects

Notes

If setClientID procedure on page 250 was not called, Null is returned.

See also

setClientID procedure on page 250, getClientID function on page 190

For more information, see the Setting and getting JMS connection and session attributes.

getConnectID function

Returns the connection ID between the JMS-provider client and broker.

Syntax

```
FUNCTION getConnectID RETURNS CHARACTER.
```

Applies to

Session objects

Notes

- This value is set by calling setConnectID procedure on page 157.
- Returns UNKNOWN when called before setConnectID procedure on page 157.

See also

setConnectID procedure on page 157

getConnectionID function

Returns the Progress OpenEdge JMS Adapter connection ID.

Syntax

```
FUNCTION getConnectionID RETURNS CHARACTER.
```

Applies to

Session objects

Notes

- This value is typically used to correlate the session to log entries on the server side.
- Returns UNKNOWN when called before beginSession procedure on page 172.

See also

For more information, see the Establishing session control.

getConnectionMetaData function

Returns a comma-separated list of connection and provider attributes in this order: JMSVersion, JMSMajorVersion, JMSMinorVersion, JMSProviderName, JMSProviderVersion, JMSProviderMajorVersion, JMSProviderMinorVersion, and AdapterVersion.

Syntax

```
FUNCTION getConnectionMetaData RETURNS CHARACTER.
```

Applies to

Session objects

See also

For more information, see the Setting and getting JMS connection and session attributes.

getDataSet function

Gets the handle to the newly created `DataSetMessage`.

Syntax

```
FUNCTION getDataSet (INPUT name, INPUT schemaLocation,
    INPUT fieldtypeMapping) RETURNS HANDLE.
```

Applies to

Message objects

Notes

- The handle parameter must be a declared handle. Any previous value of the handle parameter will be lost.
- The schema parameters specify the schema information and are passed directly to the `READ-XML` method. Specifying an `Unknown value (?)` for any of the schema parameters will result in the use of the default value for that parameter.
- The DataSet is created from reading the `DataSetMessage` and using the `READ-XML` method.
- The *name* parameter is the name of the widget-pool to be used when creating the DataSet. (For more information on widget pools, see the `CREATE-DATASET` entry in *OpenEdge Development: ABL Reference*.) A value of "?" will result in the use of the default pool.

See also

createDataSetMessage procedure on page 176, setDataSet procedure on page 251, getDataSet function on page 192

For an example, see the DataSetMessage.

getDate function

Returns a date value with no time or time zone information.

Syntax

```
FUNCTION getDate RETURNS DATE (itemName AS CHARACTER).
```

Applies to

Message objects

Notes

- Time information, if present, is removed.

- Time zone information, if present, is removed.
- If the application might receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getDateProperty function

Returns a date value with no time or time zone information.

Syntax

```
FUNCTION getDateProperty RETURNS DATE (propertyName AS CHARACTER).
```

Applies to

Message objects

Notes

- Time information, if present, is removed.
- Time zone information, if present, is removed.
- If the application might receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

See also

setDateProperty procedure on page 252, getDateProperty function on page 193

For more information, see the Accessing message properties.

getDateTime function

Returns a date-time value with no time zone information.

Syntax

```
FUNCTION getDateTime RETURNS DATETIME (itemName AS CHARACTER).
```

Applies to

Message objects

Notes

- Time zone information, if present, is removed.
- If time information is not present, the default time of 12:00AM (midnight) is added.
- If the application might receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getDateTimeProperty function

Returns a date-time property with no time zone information.

Syntax

```
FUNCTION getDateTimeProperty RETURNS DATETIME (propertyName AS CHARACTER).
```

Applies to

Message objects

Notes

- Time zone information, if present, is removed.
- If time information is not present, the default time of 12:00AM (midnight) is added.
- If the application might receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

See also

setDateTimeProperty procedure on page 254 getDateTimeProperty function on page 194

For more information, see the Accessing message properties.

getDateTime-TZ function

Returns a date-time value including time zone information.

Syntax

```
FUNCTION getDateTime-TZ RETURNS DATETIME-TZ (itemName AS CHARACTER).
```

Applies to

Message objects

Notes

- If time information is not present, the default time of 12:00AM (midnight) is added.
- If time zone information is not present, the default time zone of the client application is added.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getDateTimeTzProperty function

Returns a date-time property including time zone information.

Syntax

```
FUNCTION getDateTimeTzProperty RETURNS DATETIME-TZ (propertyName AS CHARACTER).
```

Applies to

Message objects

Notes

- If time information is not present, the default time of 12:00AM (midnight) is added.
- If time zone information is not present, the default time zone of the client application is added.

See also

setDateTimeTzProperty procedure on page 255 getDateTimeTzProperty function on page 195

For more information, see the Accessing message properties.

getDecimal function

Gets any numeric item from a `MapMessage`.

Syntax

```
FUNCTION getDecimal RETURNS DECIMAL (itemName AS CHARACTER).
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getDecimalProperty function

Returns any numeric message property.

Syntax

```
FUNCTION getDecimalProperty RETURNS DECIMAL (propertyName AS CHARACTER).
```

Applies to

Message objects

See also

For more information, see the Accessing message properties.

getDefaultPersistency function

Returns the value specified by setDefaultPersistency procedure on page 256.

Syntax

```
FUNCTION getDefaultPersistency RETURNS CHARACTER.
```

Applies to

Session objects

Notes

If setDefaultPersistency procedure on page 256 was not called, `PERSISTENT` is returned.

See also

setDefaultPersistency procedure on page 256, getDefaultPersistency function on page 197, setDefaultPriority procedure on page 256getDefaultPriority function on page 197, setDefaultTimeToLive procedure on page 257, getDefaultTimeToLive function on page 198

For more information, see the Accessing message delivery parameters.

getDefaultPriority function

Returns the value specified by setDefaultPriority procedure on page 256.

Syntax

```
FUNCTION getDefaultPriority RETURNS INTEGER.
```

Applies to

Session objects

Notes

If setDefaultPriority procedure on page 256 was not called, `4` is returned.

See also

setDefaultPersistency procedure on page 256, getDefaultPersistency function on page 197, setDefaultPriority procedure on page 256getDefaultPriority function on page 197, setDefaultTimeToLive procedure on page 257, getDefaultTimeToLive function on page 198

For more information, see the Accessing message delivery parameters.

getDefaultTimeToLive function

Returns the value specified by setDefaultTimeToLive procedure on page 257.

Syntax

```
FUNCTION getDefaultTimeToLive RETURNS DECIMAL.
```

Applies to

Session objects

Notes

If setDefaultTimeToLive procedure on page 257 was not called, `UNKNOWN` is returned.

See also

setDefaultPersistency procedure on page 256, getDefaultPersistency function on page 197, setDefaultPriority procedure on page 256getDefaultPriority function on page 197, setDefaultTimeToLive procedure on page 257, getDefaultTimeToLive function on page 198

For more information, see the Accessing message delivery parameters.

getDestinationName function

Returns the name of the destination that messages arrive from when the Message Consumer was passed to subscribe procedure on page 282 or receiveFromQueue procedure on page 240.

Syntax

```
FUNCTION getDestinationName RETURNS CHARACTER.
```

Applies to

Message Consumer objects

See also

For more information, see the Accessing message handler information.

getInt function

Gets `int`, `short`, or `byte` items from a `MapMessage`.

Syntax

```
FUNCTION getInt RETURNS INTEGER (itemName AS CHARACTER).
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt64 function on page 200, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getIntProperty function

Returns `int`, `short`, and `byte` message properties.

Syntax

```
FUNCTION getIntProperty RETURNS INTEGER (propertyName AS CHARACTER).
```

Applies to

Message objects

See also

setIntProperty procedure on page 261, getIntProperty function on page 199, setInt64Property procedure on page 262, getIntProperty function on page 199, clearProperties procedure on page 174

For more information, see the Accessing message properties.

getInt64 function

Gets INT64 items from a MapMessage.

Syntax

```
FUNCTION getInt64 RETURNS INT64 (itemName AS CHARACTER).
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setInt64 procedure on page 261, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getInt64Property function

Returns int64 message properties.

Syntax

```
FUNCTION getInt64Property RETURNS INT64 (propertyName AS CHARACTER).
```

Applies to

Message objects

See also

setInt64Property procedure on page 262, setIntProperty procedure on page 261, getIntProperty function on page 199, clearProperties procedure on page 174

For more information, see the Accessing message properties.

getItemType function

Returns the data type of an item in a MapMessage.

Syntax

```
FUNCTION getItemType RETURNS CHARACTER (itemName AS CHARACTER).
```

Applies to

Message objects

Notes

- Possible values include UNKNOWN, boolean, byte, short, char, int, datetime-tz, long, float, double, string, or longchar.
- It returns UNKNOWN if the item does not exist.
- The getItemType function cannot precisely determine certain data types. It is important to be aware of the following limitations:
  - **String values** — getItemType function on page 201 returns the longchar value for a data item consisting of a string longer than 32K. If the item is a string of 32K or less, the function returns the string value for both CHARACTER and LONGCHAR data. In the latter case, it is the responsibility of the ABL programmer to know the order of items in the MapMessage and to call the correct function to interpret the data appropriately.
  - **Date values** — getItemType function on page 201 returns the datetime-tz value for all date items.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getJMSCorrelationID function

Returns the correlation ID.

Syntax

```
FUNCTION getJMSCorrelationID RETURNS CHARACTER.
```

Applies to

Message objects

Notes

This value is an application-defined correlation ID, typically the ID of the message replied to.

See also

setJMSCorrelationID procedure on page 262, getJMSCorrelationID function on page 201

For more information, see the Accessing message header properties.

getJMSCorrelationIDAsBytes function

Returns a proprietary (JMS-provider-dependent) correlation ID.

Syntax

```
FUNCTION getJMSCorrelationIDAsBytes RETURNS RAW.
```

Applies to

Message objects

Notes

When accessing SonicMQ, the `bytesCorrelationID` field can be used for storing application-defined values.

See also

setJMSCorrelationIDAsBytes procedure on page 263 getJMSCorrelationIDAsBytes function on page 202

For more information, see the Accessing message header properties.

getJMSDeliveryMode function

Returns the delivery mode.

Syntax

```
FUNCTION getJMSDeliveryMode RETURNS CHARACTER.
```

Applies to

Message objects

Notes

- Possible values are `PERSISTENT`, `NON_PERSISTENT`, or `DISCARDABLE`.
- The message receiver never gets the `NON_PERSISTENT_ASYNC` value. A message sent using `NON_PERSISTENT_ASYNC` is received with the standard `NON_PERSISTENT` value.

See also

getJMSDestination function on page 203, getJMSRedelivered function on page 205, getMessageType function on page 214, getJMSMessageID function on page 204, getJMSDeliveryMode function on page 202, getJMSTimestamp function on page 206, getJMSExpiration function on page 203, getJMSPriority function on page 204

For more information, see the Accessing message header properties.

getJMSDestination function

Returns the name of the destination this message was sent to.

Syntax

```
FUNCTION getJMSDestination RETURNS CHARACTER.
```

Applies to

Message objects

Notes

The value is valid after the message was sent (at the sender side) and in the received message (at the receiver side).

See also

getJMSDestination function on page 203, getJMSRedelivered function on page 205, getMessageType function on page 214, getJMSMessageID function on page 204, getJMSDeliveryMode function on page 202, getJMSTimestamp function on page 206, getJMSExpiration function on page 203, getJMSPriority function on page 204

For more information, see the Accessing message header properties.

getJMSExpiration function

Returns the expiration time (GMT).

Syntax

```
FUNCTION getJMSExpiration RETURNS DECIMAL.
```

Applies to

Message objects

See also

getJMSDestination function on page 203, getJMSRedelivered function on page 205, getMessageType function on page 214, getJMSMessageID function on page 204, getJMSDeliveryMode function on page 202, getJMSTimestamp function on page 206, getJMSExpiration function on page 203, getJMSPriority function on page 204

For more information, see the Accessing message header properties.

getJMSMessageID function

Returns the message ID, a unique ID that the JMS server assigns to each message.

Syntax

```
FUNCTION getJMSMessageID RETURNS CHARACTER.
```

Applies to

Message objects

See also

getJMSDestination function on page 203, getJMSRedelivered function on page 205, getMessageType function on page 214, getJMSMessageID function on page 204, getJMSDeliveryMode function on page 202, getJMSTimestamp function on page 206, getJMSExpiration function on page 203, getJMSPriority function on page 204

For more information, see the Accessing message header properties.

getJMSPriority function

Returns priority values in the range of 0–9, where 4 is the default. Messages sent with higher priority can be expedited by the JMS-provider Broker. Priority values of 5 through 9 are expedited.

Syntax

```
FUNCTION getJMSPriority RETURNS INTEGER.
```

Applies to

Message objects

See also

getJMSDestination function on page 203, getJMSRedelivered function on page 205, getMessageType function on page 214, getJMSMessageID function on page 204, getJMSDeliveryMode function on page 202, getJMSTimestamp function on page 206, getJMSExpiration function on page 203, getJMSPriority function on page 204

For more information, see the Accessing message header properties.

getJMSRedelivered function

Returns TRUE (at the receiver side) if this is not the first delivery of this message.

Syntax

```
FUNCTION getJMSRedelivered RETURNS LOGICAL.
```

Applies to

Message objects

Notes

A second delivery can take place if the first delivery is not acknowledged by the receiver or, in a transacted session, if the transaction was rolled back.

See also

getJMSDestination function on page 203, getJMSRedelivered function on page 205, getMessageType function on page 214, getJMSMessageID function on page 204, getJMSDeliveryMode function on page 202, getJMSTimestamp function on page 206, getJMSExpiration function on page 203, getJMSPriority function on page 204

For more information, see the Accessing message header properties.

getJMSReplyTo function

Returns the reply destination.

Syntax

```
FUNCTION getJMSReplyTo RETURNS CHARACTER.
```

Applies to

Message objects

Notes

- The destination can be the name of a queue, even if the message is received from a Pub/Sub session, and the destination can be the name of a topic even if the message is received from a PTP session.
- An application must call getReplyToDestinationType function on page 219 if both a queue destination and a topic destination might be stored in the received message.

See also

setJMSReplyTo procedure on page 263, getJMSReplyTo function on page 205, hasReplyTo function on page 227, setReplyToDestinationType procedure on page 273, getReplyToDestinationType function on page 219

For more information, see the Accessing message header properties.

getJMSServerName function

Returns the value set by the preceding setJMSServerName procedure on page 264.

Syntax

```
FUNCTION getJMSServerName RETURNS CHARACTER.
```

Applies to

Session objects

Notes

If setJMSServerName procedure on page 264 is not called, The `Unknown value (?)` is returned.

See also

setJMSServerName procedure on page 264, getJMSServerName function on page 206

For more information, see the Setting and getting JMS connection and session attributes.

getJMSTimestamp function

Returns the message sending time, which is the difference, in milliseconds, between the message creation time and midnight, January 1, 1970 UTC.

Syntax

```
FUNCTION getJMSTimestamp RETURNS DECIMAL.
```

Applies to

Message objects

See also

getJMSDestination function on page 203, getJMSRedelivered function on page 205, getMessageType function on page 214, getJMSMessageID function on page 204, getJMSDeliveryMode function on page 202, getJMSTimestamp function on page 206, getJMSExpiration function on page 203, getJMSPriority function on page 204

For more information, see the Accessing message header properties.

getJMSType function

Returns a proprietary (JMS-provider-dependent) type name.

Syntax

```
FUNCTION getJMSType RETURNS CHARACTER.
```

Applies to

Message objects

Notes

When accessing SonicMQ, the `JMSType` field can be used for storing application-defined values.

See also

setJMSType procedure on page 265 getJMSType function on page 207

For more information, see the Accessing message header properties.

getLoadBalancing function

Returns a `LOGICAL` value indicating whether load balancing is enabled.

Syntax

```
FUNCTION getLoadBalancing RETURNS LOGICAL.
```

Applies to

Session objects

Notes

- TRUE indicates load balancing is enabled. FALSE indicates it is not enabled.
- With load balancing, the client is willing to have a connect request redirected to another SonicMQ Broker within a SonicMQ cluster.

See also

setLoadBalancing procedure on page 265, getLoadBalancing function on page 207

For more information, see the Load balancing.

getLogical function

Returns a `boolean` item by name from a `MapMessage`.

Syntax

```
FUNCTION getLogical RETURNS LOGICAL (itemName AS CHARACTER).
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getLogicalProperty function

Returns a `boolean` message property.

Syntax

```
FUNCTION getLogicalProperty RETURNS LOGICAL (propertyName AS CHARACTER).
```

Applies to

Message objects

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

getLongString function

Returns a `LONGCHAR` item by name from a `MapMessage`.

Syntax

```
FUNCTION getLongString RETURNS LONGCHAR (itemName AS CHARACTER).
```

Applies to

Message objects

Notes

The text is converted to the current default code page of the OpenEdge client application. To return text based on a different code page, use getLongTextCP function on page 210.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getLongStringCP function

Returns a `String` item by name from a `MapMessage` and converts the text to the specified code page.

Syntax

```
FUNCTION getLongString RETURNS LONGCHAR (code_page AS CHARACTER, itemName AS CHARACTER).
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getLongText function

Returns all text in a `TextMessage`.

Syntax

```
FUNCTION getLongText RETURNS LONGCHAR.
```

Applies to

Message objects

Notes

- Implicitly calls reset procedure on page 242.
- The text is converted to the current default code page of the OpenEdge client application. To return text based on a different code page, use getLongTextCP function on page 210.

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

getLongTextCP function

Returns all text in a `TextMessage` and converts it to the specified code page.

Syntax

```
FUNCTION getLongTextCP RETURNS LONGCHAR (code_page as CHARACTER).
```

Applies to

Message objects

Notes

- Implicitly calls reset procedure on page 242.
- The LONGCHAR data returned is converted to the code page specified by the *code_page* parameter.

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

getMapNames function

Returns a comma-separated list of the item names in a MapMessage.

Syntax

```
FUNCTION getMapNames RETURNS CHARACTER.
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

getMemptr function

Returns a reference to a MEMPTR variable that contains exactly all the bytes of a BytesMessage.

Syntax

```
FUNCTION getMemptr RETURNS MEMPTR.
```

Applies to

Message objects

Notes

- This function implicitly calls reset procedure on page 242.
- If the message was in a write-only mode, it will be in a read-only/reset mode after the call.
- getMemptr function on page 212 does not create a copy of the MEMPTR variable; it returns a reference to the data maintained by the Message object.
- deleteMessage procedure on page 182 call releases the variable's memory, and the caller must copy any data it needs or needs to modify before deleting the message.

See also

createBytesMessage procedure on page 176, setMemptr procedure on page 268, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, getContentType on page 148, getBytesCount function on page 187, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239, getMemptr function on page 212

For more information, see the see the StreamMessage and the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

getMessagePartByID function

Retrieves a message part corresponding to the given *contentID*.

Syntax

```
FUNCTION getMessagePartByID RETURNS CHARACTER
  (INPUT contentID AS INTEGER, OUTPUT messagePartH AS HANDLE).
```

Applies to

Message objects

Notes

- When you use the same handle variable to retrieve multiple message parts, after each retrieval, call deleteMessage procedure on page 182 on the handle variable to free the message part.
- The `getMessagePartByID` function returns the content-type of the message. You can use it to identify the message type of the message part.

See also

createMultipartMessage procedure on page 144, addBytesPart procedure on page 170, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getContentType on page 148, getPartCount function on page 215, getMessagePartByIndex function on page 213, getTextPartByID function on page 223, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

getMessagePartByIndex function

Retrieves a handle to the message part corresponding to the *index*.

Syntax

```
FUNCTION getMessagePartByIndex RETURNS CHARACTER
  (INPUT index AS INTEGER, OUTPUT messagePartH AS HANDLE).
```

Applies to

Message objects

Notes

- When you use the same handle variable to retrieve multiple message parts, between retrievals, call deleteMessage procedure on page 182 on the handle variable to free the message part.
- The `getMessagePartByIndex` function returns the content-type of the message. You can use it to identify the message type of the message part.

See also

createMultipartMessage procedure on page 144, addBytesPart procedure on page 170, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getPartCount function on page 215, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

getMessageType function

Returns one of the following Progress OpenEdge JMS Adapter message types: `TextMessage`, `MapMessage`, `StreamMessage`, `BytesMessage`, `HeaderMessage`, `XMLMessage`, `MultipartMessage`, `TempTableMessage`, or `DataSetMessage`.

Syntax

```
FUNCTION getMessageType RETURNS CHARACTER.
```

Applies to

Message objects

See also

getJMSDestination function on page 203, getJMSRedelivered function on page 205, getMessageType function on page 214, getJMSMessageID function on page 204, getJMSDeliveryMode function on page 202, getJMSTimestamp function on page 206, getJMSExpiration function on page 203, getJMSPriority function on page 204

For more information, see the Accessing message header properties.

getNoAcknowledge function

Returns `TRUE` if setNoAcknowledge procedure on page 269 was called.

Syntax

```
FUNCTION getNoAcknowledge RETURNS LOGICAL.
```

Applies to

Message Consumer objects

See also

acknowledgeAndForward procedure on page 141, setSingleMessageAcknowledgement procedure on page 169, getSingleMessageAcknowledgement function on page 155, setNoAcknowledge procedure on page 269, getNoAcknowledge function on page 214

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.

getPartCount function

Returns the number of parts in a `MultipartMessage`.

Syntax

```
FUNCTION getPartCount RETURNS INTEGER.
```

Applies to

Message objects

See also

createMultipartMessage procedure on page 144, addBytesPart procedure on page 170, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getContentType on page 148, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

getPassword

Returns the value set by the preceding setPassword procedure on page 270.

For more information, see the MultipartMessage and the MultiPartMessage example.getPassword function

Syntax

```
FUNCTION getPassword RETURNS CHARACTER.
```

Applies to

Session objects

Note

If setPassword procedure on page 270 was not called, The `Unknown value (?)` is returned.

See also

setPassword procedure on page 270

For more information, see the Setting and getting JMS connection and session attributes.

getProcHandle function

Returns the handle to a procedure that contains the name of an internal procedure for handling messages.

Syntax

```
FUNCTION getProcHandle RETURNS HANDLE.
```

Applies to

Message Consumer objects

See also

getProcName function on page 216, getProcHandle function on page 216

For more information, see the Accessing message handler information.

getProcName function

Returns the name of the internal procedure for handling messages.

Syntax

```
FUNCTION getProcName RETURNS CHARACTER.
```

Applies to

Message Consumer objects

See also

getProcName function on page 216, getProcHandle function on page 216

For more information, see the Accessing message handler information.

getPropertyNames function

Returns a comma-separated list of the properties of a message.

Syntax

```
FUNCTION getPropertyNames RETURNS CHARACTER.
```

Applies to

Message objects

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

getPropertyType function

Returns the message property's data type.

Syntax

```
FUNCTION getPropertyType RETURNS CHARACTER (propertyName AS CHARACTER).
```

Parameter

propertyName

The message property's data type. Possible values are: `UNKNOWN`, `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`, or `string`.

Applies to

Message objects

Notes

- If the property was not set in the message, the `UNKNOWN` is returned.
- Since date-time values are transmitted as `String` data, the function cannot distinguish them from other strings. The ABL programmer must know the order of properties in the header and call the correct function to interpret date-time values appropriately.

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

getReplyAutoDelete function

Returns whether all reply messages are to be automatically deleted or not.

Syntax

```
FUNCTION getReplyAutoDelete RETURNS LOGICAL.
```

Applies to

Message Consumer objects

See also

setReplyPersistency procedure on page 271, getReplyPersistency function on page 218, setReplyPriority procedure on page 272, getReplyPriority function on page 218, setReplyTimeToLive procedure on page 272, getReplyTimeToLive function on page 219, setReplyAutoDelete procedure on page 271, getReplyAutoDelete function on page 217

For more information, see the Accessing message handler information and the Setting reply properties.

getReplyPersistency function

Returns the value for message persistency.

Syntax

```
FUNCTION getReplyPersistency RETURNS CHARACTER.
```

Applies to

Message Consumer objects

Notes

If setReplyPersistency procedure on page 271 was not called, `PERSISTENT` is returned.

See also

setReplyPersistency procedure on page 271, getReplyPersistency function on page 218, setReplyPriority procedure on page 272, getReplyPriority function on page 218, setReplyTimeToLive procedure on page 272, getReplyTimeToLive function on page 219, setReplyAutoDelete procedure on page 271, getReplyAutoDelete function on page 217

For more information, see the Accessing message handler information and the Setting reply properties.

getReplyPriority function

Returns the priority of the reply messages.

Syntax

```
FUNCTION getReplyPriority RETURNS INTEGER.
```

Applies to

Message Consumer objects

Notes

If setReplyPriority procedure on page 272 was not called, the returned value is 4.

See also

setReplyPersistency procedure on page 271, getReplyPersistency function on page 218, setReplyPriority procedure on page 272, getReplyPriority function on page 218, setReplyTimeToLive procedure on page 272, getReplyTimeToLive function on page 219, setReplyAutoDelete procedure on page 271, getReplyAutoDelete function on page 217

For more information, see the Accessing message handler information and the Setting reply properties.

getReplyTimeToLive function

Returns the time to live value (in milliseconds) of the reply messages.

Syntax

```
FUNCTION getReplyTimeToLive RETURNS DECIMAL.
```

Applies to

Message Consumer objects

Notes

If setReplyTimeToLive procedure on page 272 was not called, `UNKNOWN` is returned.

See also

setReplyPersistency procedure on page 271, getReplyPersistency function on page 218, setReplyPriority procedure on page 272, getReplyPriority function on page 218, setReplyTimeToLive procedure on page 272, getReplyTimeToLive function on page 219, setReplyAutoDelete procedure on page 271, getReplyAutoDelete function on page 217

For more information, see the Accessing message handler information and the Setting reply properties.

getReplyToDestinationType function

Returns `queue`, `topic`, or `UNKNOWN`.

Syntax

```
FUNCTION getReplyToDestinationType RETURNS CHARACTER.
```

Applies to

Message objects

Notes

- Applications use this function when the domain of the `ReplyTo` field is not known.
- The *type* value can be `queue` or `topic`.
- If getReplyToDestinationType function on page 219 is not called, a default type is automatically set when the message is sent, according to the type of the session: `queue` for PTP or `topic` for Pub/Sub.
- For a `jmsSession`, the default is queue.

See also

setJMSReplyTo procedure on page 263, getJMSReplyTo function on page 205, hasReplyTo function on page 227, setReplyToDestinationType procedure on page 273, getReplyToDestinationType function on page 219

For more information, see the Accessing message header properties.

getReuseMessage function

Returns the value set by setReuseMessage procedure on page 274.

Syntax

```
FUNCTION getReuseMessage RETURNS LOGICAL.
```

Applies to

Message Consumer objects

Notes

`TRUE` if setReuseMessage procedure on page 274 was called; if not, it returns `FALSE`.

See also

setReuseMessage procedure on page 274, getReuseMessage function on page 220

For more information, see the Reusing messages.

getSaxWriter function

Creates a SAX-WRITER and sets the output destination to an internal longchar for intermediate storage.

Syntax

```
FUNCTION getSaxWriter (INPUT name) RETURNS HANDLE.
```

Applies to

Message objects

Notes

- The caller uses SAX-WRITER methods on the returned handle to create the XML document. (which will be written to the internal longchar).
- The *name* parameter is the name of the widget-pool to be used when creating the SAX-WRITER. The Unknown value (?) results in the use of the default pool.
- When XML creation is completed, call deleteSaxWriter procedure on page 183.
- Possible errors are returned by CREATE-SAX-WRITER or SET-OUTPUT-DESTINATION.

See also

createXMLMessage procedure on page 181, setX-Document procedure on page 280, getX-Document function on page 226, setSaxReader procedure on page 274, getSaxWriter function on page 221, deleteSaxWriter procedure on page 183

For more information, see the XMLMessage.

getSession function

Returns a handle to the session.

Syntax

```
FUNCTION getSession RETURNS HANDLE.
```

Applies to

Message Consumer objects

See also

beginSession procedure on page 172, getSession function on page 221, deleteSession procedure on page 183

For more information, see the Accessing message handler information.

getShutdownWaitFor function

Returns a LOGICAL value indicating the current value of the shutdown WAIT-FOR flag.

Syntax

```
FUNCTION getShutdownWaitFor RETURNS LOGICAL.
```

Applies to

Session objects

Notes

- getShutdownWaitFor function on page 222 returns YES if setShutdownWaitFor procedure on page 276 was never called.

See also

setShutdownWaitFor procedure on page 276

getTempTable function

Gets the handle to the newly created TempTable.

Syntax

```
FUNCTION getTempTable (INPUT name, INPUT schemaLocation,
    INPUT fieldtypeMapping) RETURNS HANDLE.
```

Applies to

Message objects

Notes

- The handle parameter must be a declared handle. Any previous value of the handle parameter will be lost.
- The schema parameters specify the schema information and are passed directly to the READ-XML method. Specifying an Unknown value (?) for any of the schema parameters will result in the use of the default value for that parameter.
- The TempTable is created from reading the TempTableMessage and using the READ-XML method.
- The *name* parameter is the name of the widget-pool to be used when creating the TempTable. (For more information on widget pools, see the CREATE-TEMP-TABLE entry in *OpenEdge Development: ABL Reference*.) A value of "?" will result in the use of the default pool.

See also

createTempTableMessage procedure on page 180, setTempTable procedure on page 278, getTempTable function on page 222

For more information, see the TempTableMessage.

getText function

Returns all text in a TextMessage or XMLMessage.

Syntax

```
FUNCTION getText RETURNS CHARACTER.
```

Applies to

Message objects

Notes

- A run-time error occurs if the message is too large to be handled by the ABL interpreter.
- Implicitly calls reset procedure on page 242.

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

getTextPartByID function

Retrieves a text part and returns the content type as a CHARACTER string.

Syntax

```
FUNCTION getTextPartByID RETURNS CHARACTER
  (INPUT contentID AS INTEGER, OUTPUT partBody AS CHARACTER).
```

Applies to

Message objects

Notes

- This function converts the text part from UTF-8 to the SESSION:CPINTERNAL code page.
- If the message body exceeds 32K, this function raises an error. To avoid this, use getBytesPartByID function on page 145.

See also

getBytesPartByID function on page 145, getMessagePartByID function on page 212, getTextPartByID function on page 223

For more information, see MultipartMessage

getTextPartByIndex function

Retrieves a text part and returns the content type as a CHARACTER string.

Syntax

```
FUNCTION getTextPartByIndex RETURNS CHARACTER
  (INPUT iIndex AS INTEGER, OUTPUT partBody AS CHARACTER).
```

Applies to

Message objects

Notes

- This function converts the text part from UTF-8 to the SESSION:CPINTERNAL code page.
- If the message body exceeds 32K, this function raises an error. To avoid this, use getBytesPartByIndex function on page 145.

See also

getBytesPartByIndex function on page 145, getMessagePartByIndex function on page 213, getTextPartByIndex function on page 224

For more information, see the MultipartMessage.

getTextSegment function

Returns the next text segment when handling large messages in read-only mode.

Syntax

```
FUNCTION getTextSegment RETURNS CHARACTER.
```

Applies to

Message objects

Notes

As an alternative to retrieving multiple text segments with getTextSegment function on page 224, you can use getLongText function on page 210 to retrieve LONGCHAR data in a single operation.

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

getTransactedReceive function

Returns the value set by the preceding setTransactedReceive procedure on page 279.

Syntax

```
FUNCTION getTransactedReceive RETURNS LOGICAL.
```

Applies to

Session objects

Notes

If setTransactedReceive procedure on page 279 was not called, FALSE is returned.

See also

setTransactedReceive procedure on page 279, getTransactedReceive function on page 225, setTransactedSend procedure on page 279, getTransactedSend function on page 225

For more information, see the Setting and getting JMS connection and session attributes.

getTransactedSend function

Returns the value set by the preceding setTransactedSend procedure on page 279.

Syntax

```
FUNCTION getTransactedSend RETURNS LOGICAL.
```

Applies to

Session objects

Notes

If setTransactedSend procedure on page 279 was not called, FALSE is returned.

See also

For more information, see the Setting and getting JMS connection and session attributes.

getUser function

Returns the value set by the preceding setUser procedure on page 280.

Syntax

```
FUNCTION getUser RETURNS CHARACTER.
```

Applies to

Session objects

Notes

If setUser procedure on page 280 was not called, the `Unknown value (?)` is returned.

See also

setUser procedure on page 280, getUser function on page 226

For more information, see the Setting and getting JMS connection and session attributes.

getX-Document function

Parses an XML document from a SonicMQ `XMLMessage` into an `X-DOCUMENT`.

Syntax

```
FUNCTION getX-Document RETURNS HANDLE.
```

Applies to

Message objects

Notes

- The handle to the created X-DOCUMENT is returned as the value of this function. CREATE X-DOCUMENT can return an error.
- The caller is responsible for deleting the handle when the application is done processing the X-DOCUMENT.
- Errors from getLongText function on page 210 and xdoc:LOAD will be returned.

See also

createXMLMessage procedure on page 181, setX-Document procedure on page 280, getX-Document function on page 226, setSaxReader procedure on page 274, getSaxWriter function on page 221, deleteSaxWriter procedure on page 183

For more information, see the XMLMessage.

hasReplyTo function

Returns TRUE if the JMSreplyTo header was set.

Syntax

```
FUNCTION hasReplyTo RETURNS LOGICAL.
```

Applies to

Message objects

See also

setJMSReplyTo procedure on page 263, getJMSReplyTo function on page 205, hasReplyTo function on page 227, setReplyToDestinationType procedure on page 273, getReplyToDestinationType function on page 219

For more information, see the Accessing message header properties.

inErrorHandling function

Returns TRUE when called from a message handler if the application is handling an error message.

Syntax

```
FUNCTION inErrorHandling RETURNS LOGICAL.
```

Applies to

Message Consumer objects

See also

inErrorHandling function on page 227, inMessageHandling function on page 228, inQueueBrowsing function on page 228, inReplyHandling function on page 229

For more information, see the Accessing message handler information.

inMessageHandling function

Returns TRUE when called from a message handler if the application is handling the data in a subscription (or queue) message.

Syntax

```
FUNCTION inMessageHandling RETURNS LOGICAL.
```

Applies to

Message Consumer objects

See also

inErrorHandling function on page 227, inMessageHandling function on page 228, inQueueBrowsing function on page 228, inReplyHandling function on page 229

For more information, see the Accessing message handler information.

inQueueBrowsing function

Returns TRUE when called from a message handler if an application is handling a queue browsing message.

Syntax

```
FUNCTION inQueueBrowsing RETURNS LOGICAL.
```

Applies to

Message Consumer objects

See also

inErrorHandling function on page 227, inMessageHandling function on page 228, inQueueBrowsing function on page 228, inReplyHandling function on page 229

For more information, see the Accessing message handler information.

inReplyHandling function

Returns TRUE when called from a message handler if an application is handling a reply message.

Syntax

```
FUNCTION inReplyHandling RETURNS LOGICAL.
```

Applies to

Message Consumer objects

See also

inErrorHandling function on page 227, inMessageHandling function on page 228, inQueueBrowsing function on page 228, inReplyHandling function on page 229

For more information, see the Accessing message handler information.

isMessagePart function

Returns TRUE if the part specified by *index* is a SonicMQ message.

Syntax

```
FUNCTION isMessagePart RETURNS LOGICAL (INPUT index AS INTEGER).
```

Applies to

Message objects

Notes

If isMessagePart function on page 229 returns TRUE, use message-part access methods.

See also

createMultipartMessage procedure on page 144, addBytesPart procedure on page 170, addMessagePart procedure on page 142, addTextPart procedure on page 170, isMessagePart function on page 229, getContentType on page 148, getPartCount function on page 215, getMessagePartByID function on page 212, getMessagePartByIndex function on page 213, writeBytesFromRaw procedure on page 286, readBytesToRaw procedure on page 233, setMemptr procedure on page 268, getMemptr function on page 212

For more information, see the MultipartMessage and the MultiPartMessage example.

JMS-MAXIMUM-MESSAGES global variable

Changes the maximum number of JMS messages in an OpenEdge session.

Syntax

```
DEFINE NEW GLOBAL SHARED VAR JMS-MAXIMUM-MESSAGES AS INTEGER INIT new-val.
```

Notes

- The total number of messages includes messages created by the application and messages received from JMS.
- The default is 50.
- If you exceed the message limit, an error is returned.
- To change the default to *new–val*, the variable definition must be included in the main procedure of the OpenEdge application.

See also

For information on this global variable in context, see the Setting the maximum number of messages.

messageHandler procedure

Handles incoming JMS and error messages.

Syntax

```
PROCEDURE messageHandler.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
DEFINE OUTPUT PARAMETER reply AS HANDLE.
```

Parameters

message

The message.

messageConsumer

The Message Consumer object that contains this message handler. The application can use the Message Consumer object to get context information about the message (for example, the session handle to the session that received that message) and the context (for example, the session handler).

reply

A handle to the reply message, if any. The application can reply to the message automatically without having to extract the `reply to` fields. The application can set the `reply` parameter with a reply message, which is automatically sent to the `JMSReplyTo` destination of the message. If the setReplyAutoDelete procedure on page 271`(true)` Message Consumer procedure is called, the reply message is automatically deleted after being sent.

Applies to

Message objects

Notes

- The message handler is written by an application and must be registered with a Message Consumer object.
- When a message is received, the message handler is called automatically so the application can process the message.

See also

createMessageConsumer procedure on page 178, deleteConsumer procedure on page 182, messageHandler procedure on page 230, waitForMessages procedure on page 283

For more information see the Message Consumer objects, the Consuming messages, the Terminating the Message Consumer object, and the Creating a message handler process.

moveToNext procedure

Moves the cursor to the next data item in a `StreamMessage` and returns its data type.

Syntax

```
FUNCTION moveToNext RETURNS CHARACTER.
```

Applies to

Message objects

Notes

- Possible return values include the `Unknown value (?)`, `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `string`, or `byte`.
- The `Unknown value (?)`is returned when the value of the item is `NULL`.
- When the message is received or after reset procedure on page 242 is called, the cursor is set before the first data item.
- It is an error to try to move the cursor beyond the last item.
- moveToNext procedure on page 231 function cannot precisely determine certain data types. It is important to be aware of the following limitations:

- **String values** — moveToNext procedure on page 231 returns the `longchar` value for a data item consisting of a string longer than 32K. If the item is a string of 32K or less, the function returns the `string` value for both `CHARACTER` and `LONGCHAR` data. In the latter case, it is the responsibility of the ABL programmer to know the order of items in the `StreamMessage` and to call the correct function to interpret the data appropriately.
- **Date values** — moveToNext procedure on page 231 returns the `string` value for all date items. The ABL programmer must know the order of items in the `StreamMessage` and call the correct function to interpret the data appropriately.
- **INT64 values** — moveToNext procedure on page 231 returns the `long` value for `INT64` message data. It is the responsibility of the ABL programmer to know the order of items in the `StreamMessage` and to call the correct function to interpret the data appropriately.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readInt64 function on page 238, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

publish procedure

Publishes a message to a topic.

Syntax

```
PROCEDURE publish.
DEFINE INPUT PARAMETER topicName AS CHARACTER.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INTEGER.
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.
```

Parameters

topicName

The topic to which the message is published.

message

The Message object.

priority

The message priority (optional): 0–9. Session default is used if UNKNOWN.

timeToLive

Time to live, in milliseconds (optional). Session default is used if UNKNOWN.

deliveryMode

The delivery mode (optional): PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, DISCARDABLE, or UNKNOWN (?). Session default is used if UNKNOWN.

Applies to

Session objects

Notes

- If the publication is in reply to a received message, *topicName* can be the ReplyTo field obtained from the original message.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

cancelDurableSubscription procedure on page 173, publish procedure on page 232, subscribe procedure on page 282

For more information, see the Publishing messages to a topic, the Subscribing to a topic, the Durable subscriptions, and the Methods unique to Pub/Sub messaging.

For an example, see the Pub/Sub messaging example on page 85.

readBytesToRaw procedure

Returns byte array data from the body of a StreamMessage or a BytesMessage.

Syntax

```
FUNCTION readBytesToRaw RETURNS RAW.
```

Applies to

Message objects

Notes

- It can be called in read-only mode to return the next byte segment in a BytesMessage.
- The size of all the byte segments other than the last one is 8192; the size of the last one is 8192 or less.

See also

createStreamMessage procedure on page 179, getContentType on page 148, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage and the BytesMessage.

readChar function

Returns any message data segment except bytes data from the body of a `StreamMessage`.

Syntax

```
FUNCTION readChar RETURNS CHARACTER.
```

Applies to

Message objects

Notes

The size of all the character segments other than the last one is 8192; the size of the last one is 8192 or less.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readDate function

Returns a `DATE` value with no time zone information from the body of a `StreamMessage`.

Syntax

```
FUNCTION readDate RETURNS DATE.
```

Applies to

Message objects

Notes

- Time information, if present, is removed.
- Time zone information, if present, is removed.
- If the application might receive messages originating in different time zones, the ABL programmer should ensure that date values are interpreted correctly.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readDateTime function

Returns a `DATETIME` value with no time zone information from the body of a `StreamMessage`.

Syntax

```
FUNCTION readDateTime RETURNS DATETIME.
```

Applies to

Message objects

Notes

- Time zone information, if present, is removed.
- If time information is not present, the default time of 12:00AM (midnight) is added.
- If the application may receive messages originating in different time zones, the ABL programmer should ensure that date/time values are interpreted correctly.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readDateTime-TZ function

Returns a `DATETIME-TZ` value from the body of a `StreamMessage`.

Syntax

```
FUNCTION readDateTimeTz RETURNS DATETIME-TZ.
```

Applies to

Message objects

Notes

- If time information is not present, the default time of 12:00AM (midnight) is added.
- If time zone information is not present, the default time zone of the client application is added.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readDecimal function

Returns any numeric data from the body of a `StreamMessage`.

Syntax

```
FUNCTION readDecimal RETURNS DECIMAL.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readInt function

Returns `int`, `short`, or `byte` data from the body of a `StreamMessage`.

Syntax

```
FUNCTION readInt RETURNS INTEGER.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeInt64 procedure on page 291, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt64 function on page 238, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readInt64 function

Returns INT64 data from the body of a StreamMessage.

Syntax

```
FUNCTION readInt64 RETURNS INT64.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeInt64 procedure on page 291, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readLogical function

Returns boolean data from the body of a StreamMessage.

Syntax

```
FUNCTION readLogical RETURNS LOGICAL.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readLongString function

Returns `String` data from the body of a `StreamMessage`.

Syntax

```
FUNCTION readLongString RETURNS LONGCHAR.
```

Applies to

Message objects

Notes

- The text is converted to the current default code page of the OpenEdge client application.
- To return text based on a different code page, use readLongStringCP function on page 239.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

readLongStringCP function

Returns `LONGCHAR` data from the body of a `StreamMessage` and converts it to the specified code page.

Syntax

```
FUNCTION readLongStringCP RETURNS LONGCHAR (code_page as CHARACTER).
```

Applies to

Message objects

Notes

The LONGCHAR data returned is converted to the code page specified by the *code_page* parameter.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

receiveFromQueue procedure

Receives messages from a queue.

Syntax

```
PROCEDURE receiveFromQueue.
DEFINE INPUT PARAMETER queueName AS CHARACTER.
DEFINE INPUT PARAMETER messageSelector AS CHARACTER.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

Parameters

queueName

The queue from which the messages are received.

messageSelector

A message selector. If UNKNOWN, receives all messages.

messageConsumer

A Message Consumer object, which handles the messages asynchronously.

Applies to

Session objects

Notes

- The messages are handled asynchronously by the *messageConsumer* procedure.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

browseQueue procedure on page 172, receiveFromQueue procedure on page 240, sendToQueue procedure on page 244

For more information, see theSending messages to a queue, the Receiving messages from a queue, and the Methods unique to Point-to-Point messaging.

For an example, see the PTP message example on page 82.

recover procedure

Redelivers all unacknowledged messages received up to that point in the current session.

Syntax

```
PROCEDURE recover.
```

Applies to

Session objects

Notes

- It is an error to call this method if the session is transacted for receiving. Call the rollbackReceive procedure on page 243 instead.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

commitSend procedure on page 175, commitReceive procedure on page 175. rollbackSend procedure on page 243, rollbackReceive procedure on page 243, recover procedure on page 241

For more information, see the Transaction and recovery procedures.

requestReply procedure

Creates a temporary queue or topic and sets the `JMSReplyTo` message header field. Then `requestReply procedure` sends the message to the destination specified and designates the *messageConsumer* parameter for processing replies.

Syntax

```
PROCEDURE requestReply.
DEFINE INPUT PARAMETER destination AS CHARACTER.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER replySelector AS CHARACTER.      /*UNKNOWN means
                                                        receiving all replies*/
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.  /*UNKNOWN is illegal*/
DEFINE INPUT PARAMETER priority AS INTEGER.           /*Session default is
                                                        used if UNKNOWN./*
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.      /*Session default is
                                                        used if UNKNOWN./*
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.       /*Session default is
                                                        used if UNKNOWN./*
```

Applies to

Session objects

Notes

- The term *destination* is used for both topics and queues.
- The ABL–JMS implementation automates the request/reply sequence:
  - Sending a reply by setting the reply `OUTPUT` parameter of the message handler
  - Requesting a reply by calling requestReply procedure on page 241 with a reply Message Consumer
- The ABL–JMS implementation uses a temporary destination for the reply. It is an error to set the `JMSReplyTo` field of the message explicitly if `requestReply` is used. The reply is received by messageConsumer asynchronously, just like any other message reception. The temporary destination is deleted when the Message Consumer object is deleted.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

For more information, see the Request/Reply.

reset procedure

Changes the mode of a message from write-only to read-only mode and positions the cursor before the first data item.

Syntax

```
PROCEDURE reset.
```

Applies to

Message objects

Notes

- Sending the message causes an implicit call to reset procedure on page 242.
- The message becomes read-only and arrives at the receiver in a reset state.

See also

For more information, see the TextMessage, the StreamMessage, and the BytesMessage.

rollbackReceive procedure

Starts redelivering the messages received up to that point in the current transaction.

Syntax

```
PROCEDURE rollbackReceive.
```

Applies to

Session objects

Notes

- Redelivers messages that have been received, but not acknowledged.
- It is an error to call this procedure in a Session object that is not transacted for receiving.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

commitSend procedure on page 175, commitReceive procedure on page 175. rollbackSend procedure on page 243, rollbackReceive procedure on page 243, recover procedure on page 241

For more information, see the Transaction and recovery procedures.

rollbackSend procedure

Discards all messages sent up to that point in the current transaction.

Syntax

```
PROCEDURE rollbackSend.
```

Applies to

Session objects

Notes

- It is an error to call this method in a Session object that is not transacted for sending.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

commitSend procedure on page 175, commitReceive procedure on page 175. rollbackSend procedure on page 243, rollbackReceive procedure on page 243, recover procedure on page 241

For more information, see the Message acknowledgement, forwarding, and recovery.

sendToQueue procedure

Sends a message to a queue.

Syntax

```
PROCEDURE sendToQueue.
DEFINE INPUT PARAMETER queueName AS CHARACTER.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INTEGER.
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.
```

Parameters

queueName

The queue to which the message is sent.

message

The Message object.

priority

The message priority (optional): 0–9. If `Unknown value (?)`, the session default is used.

timeToLive

Time to live, in milliseconds (optional). If `Unknown value (?)`, the session default is used.

deliveryMode

The delivery mode (optional): `PERSISTENT`, `NON_PERSISTENT`, `NON_PERSISTENT_ASYNC`, or `UNKNOWN` (?). If `Unknown value (?)`, the session default is used.

Applies to

Session objects

Notes

- If the sending is in reply to a received message, *queueName* can be the `ReplyTo` field obtained from the original message.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

browseQueue procedure on page 172, receiveFromQueue procedure on page 240, sendToQueue procedure on page 244

For more information, see theSending messages to a queue, the Receiving messages from a queue, and the Methods unique to Point-to-Point messaging.

For an example, see the PTP message example on page 82.

setAdapterService procedure

Specifies the service name under which the Progress OpenEdge JMS Adapter BrokerConnect is registered with.

Syntax

```
PROCEDURE setAdapterService.
DEFINE INPUT PARAMTER serviceName AS CHARACTER,
```

Applies to

Session objects (for BrokerConnect only)

Notes

- The default is adapter.progress.jms.
- If the Progress OpenEdge JMS Adapter uses adapter.progress.jms, calling setAdapterService procedure on page 245 is unnecessary.
- If the application uses a `-URL` parameter to connect to the Progress OpenEdge JMS Adapter, that parameter includes the service name; any subsequent calls to `setAdapterService` are ignored.

See also

setAdapterService procedure on page 245, getAdapterService function on page 186

For more information, see the Setting and getting JMS connection and session attributes.

setApplicationContext procedure

Passes context to the message handler.

Syntax

```
PROCEDURE setApplicationContext.
DEFINE INPUT PARAMETER handler AS HANDLE.
```

Applies to

Message Consumer objects

Notes

- The *handler* parameter is typically a handle to a persistent procedure implemented by the application.
- When the message handler is called, it gets that handler and uses it, for example, to deposit error information in the application's context by calling a specific handler's internal procedure.

See also

setApplicationContext procedure on page 245, getApplicationContext function on page 186

For more information, see the Accessing message handler information and the Creating a message handler process.

setBoolean procedure

Converts data to the JMS `boolean` data type in a `MapMessage`.

Syntax

```
PROCEDURE setBoolean.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

Applies to

Message objects

Notes

An `Unknown value (?)` is considered `FALSE`.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setBooleanProperty procedure

Sets a `boolean` message property.

Syntax

```
PROCEDURE setBooleanProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS LOGICAL.
```

Applies to

Message objects

Notes

An `Unknown value (?)` is considered a `FALSE` value.

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

setBrokerURL procedure

Sets the value of the JMS-provider Broker URL.

Syntax

```
PROCEDURE setBrokerURL.
DEFINE INPUT PARAMETER brokerURL AS CHARACTER.
```

Parameter

brokerURL

The URL for the JMS-provider Broker.

Applies to

Session objects

Notes

- If set on the client, it overwrites the default broker URL property set on the Progress OpenEdge JMS Adapter side.
- The creation of a session fails if no value is set on the client or at the Progress OpenEdge JMS Adapter.

See also

setBrokerURL procedure on page 247, getBrokerURL function on page 187

For more information, see the Setting and getting JMS connection and session attributes.

setByte procedure

Converts data in a `MapMessage` to the JMS `byte` data type.

Syntax

```
PROCEDURE setByte.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INTEGER.
```

Applies to

Message objects

Notes

- Byte values range from –128 to 127.
- The server returns a `NumberFormatException` message for a value overflow. For example, calling `setByte("item1", 1000)` results in a value overflow.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setByteProperty procedure

Sets a `byte` property in a message; the values range from –128 to 127.

Syntax

```
PROCEDURE setByteProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INTEGER.
```

Applies to

Message objects

Notes

The server returns a NumberFormatException message for a value overflow. For example, calling `setByteProperty("prop1", 1000)` results in a value overflow.

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

setBytesFromRaw procedure

Converts data in a `MapMessage` to the JMS `byte` data type.

Syntax

```
PROCEDURE setBytesFromRaw.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMTER values AS RAW.
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setChar procedure

Converts data in a `MapMessage` to the JMS `char` data type.

Syntax

```
PROCEDURE setChar.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

Applies to

Message objects

Notes

The number of characters in the `char` value must be one.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setClientID procedure

Sets the client ID value for the JMS-provider Broker connection and overwrites the default client ID set on the server side.

Syntax

```
PROCEDURE setClientID.
DEFINE INPUT PARAMETER clientID AS CHARACTER.
```

Parameter

clientID

Client ID value for the JMS-provider Broker connection.

Applies to

Session objects

Notes

- A client ID is required for durable subscriptions and for client persistence.
- If called, setClientID procedure on page 250 overwrites the default client ID set on the server side.

See also

setClientID procedure on page 250, getClientID function on page 190

For more information, see the Setting and getting JMS connection and session attributes, the Methods unique to Pub/Sub messaging, and the Subscribing to a topic.

setDataSet procedure

Sets the `DataSetMessage`.

Syntax

```
PROCEDURE setDataSet.
DEFINE INPUT PARAMETER dsHdl AS HANDLE.
DEFINE INPUT PARAMETER schemaLocation AS CHARACTER.
DEFINE INPUT PARAMETER writeSchema AS LOGICAL.
```

Applies to

Message objects

Notes

- The handle parameter must be a handle to a valid DataSet.
- The schema parameters specify the schema information and are passed directly to the `WRITE-XML` method. Specifying an `Unknown value (?)` for any of the schema parameters will result in the use of the default value for that parameter.
- The `DataSetMessage` is converted to an `XMLMessage` using the `WRITE-XML` method and setLongText procedure on page 267.

See also

createDataSetMessage procedure on page 176, setDataSet procedure on page 251, getDataSet function on page 192

For an example, see the DataSetMessage.

setDate procedure

Sets a date value as `String` data in a `MapMessage`.

Syntax

```
PROCEDURE setDate.
DEFINE INPUT PARAMETER itemname as CHARACTER.
DEFINE INPUT PARAMETER value as DATE.
```

Applies to

Message objects

Notes

- The procedure writes *value* as a DATETIME-TZ value, adding default time and time zone information:
  - A time of 12:00AM (midnight)
  - The default time zone of the client application
- If the message might be consumed by an application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly.
- The format of the DATETIME-TZ value is a subset of the ISO8601 format.
- An error is returned if the Unknown value (?) is specified.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setDateProperty procedure

Sets a date property in a message header.

Syntax

```
PROCEDURE setDateProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DATE.
```

Applies to

Message objects

Notes

- The date value is set as a `String` message property.
- The procedure writes *`propertyValue`* as a `DATETIME-TZ` value, adding default time and time zone information:
  - A time of 12:00AM (midnight)
  - The default time zone of the client application
- The format of the `DATETIME-TZ` value is a subset of the ISO8601 format.
- An error is returned if the `Unknown value (?)` is specified.
- If the message might be consumed by a non-OpenEdge application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly. OpenEdge applications that receive the message correctly interpret the `DATETIME-TZ` value.

See also

setDateProperty procedure on page 252, getDateProperty function on page 193

For more information, see the Accessing message properties.

setDateTime procedure

Sets a date-time value as `String` data in a `MapMessage`.

Syntax

```
PROCEDURE setDateTime.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS DATETIME.
```

Applies to

Message objects

Notes

- The date-time value is set as a `String` message property.
- The procedure writes *`value`* as a `DATETIME-TZ` value, adding default time zone information. The format of the `DATETIME-TZ` value is a subset of the ISO8601 format.
- If the message might be consumed by a non-OpenEdge application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly. OpenEdge applications that receive the message correctly interpret the `DATETIME-TZ` value.
- An error is returned if the `Unknown value (?)` is specified.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setDateTimeProperty procedure

Sets a date-time value.

Syntax

```
PROCEDURE setDateTimeProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DATETIME.
```

Applies to

Message objects

Notes

- The date-time value is set as a `String` property in a message header.
- The procedure writes *value* as a `DATETIME-TZ` value, adding default time zone information.

See also

setDateTimeProperty procedure on page 254, getDateTimeProperty function on page 194

For more information, see the Accessing message properties.

setDateTime-TZ procedure

Sets a date-time value, including time zone information in a `MapMessage`.

Syntax

```
PROCEDURE setDateTime-TZ.
DEFINE INPUT PARAMETER itemName as CHARACTER.
DEFINE INPUT PARAMETER value as DATETIME-TZ
```

Applies to

Message objects

Notes

- The date-time value, including time zone information, is set as a `String` message property.
- An error is returned if the `Unknown value (?)` is specified.
- If the message might be consumed by a non-OpenEdge application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly. OpenEdge applications that receive the message correctly interpret the `DATETIME-TZ` value.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setDateTimeTzProperty procedure

Sets a date-time value, including time zone information.

Syntax

```
PROCEDURE setDateTimeTzProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DATETIME-TZ.
```

Applies to

Message objects

Notes

- The date-time value, including time zone information, is set as a `String` message property in the message header.
- The procedure writes *value* as a `DATETIME-TZ` value, adding default time and time zone information.

See also

setDateTimeTzProperty procedure on page 255, getDateTimeTzProperty function on page 195

For more information, see the Accessing message properties.

setDefaultPersistency procedure

Sets the default message persistency value for all messages sent in that session.

Syntax

```
PROCEDURE setDefaultPersistency.
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.
```

Applies to

Session objects

Notes

- Possible values include: `PERSISTENT`, `NON_PERSISTENT`, `NON_PERSISTENT_ASYNC`, `DISCARDABLE`, and `UNKNOWN` (?).
- The default value is `PERSISTENT`.
- A call with an `Unknown value (?)` has no effect.
- Use `DISCARDABLE` only when publishing to a topic. If `DISCARDABLE` is used when publishing other than to a topic, an error is raised.
- The evaluation is case-insensitive.
- `NON_PERSISTENT_ASYNC` is a SonicMQ extension of the JMS specification.

See also

setDefaultPersistency procedure on page 256, getDefaultPersistency function on page 197, setDefaultPriority procedure on page 256getDefaultPriority function on page 197, setDefaultTimeToLive procedure on page 257, getDefaultTimeToLive function on page 198

For more information, see the Accessing message delivery parameters.

setDefaultPriority procedure

Sets the default message priority for all messages sent in that session.

Syntax

```
PROCEDURE setDefaultPriority.
DEFINE INPUT PARAMETER priority AS INTEGER.
```

Applies to

Session objects

Notes

- The range of priority values is 0–9. The default is 4.
- Setting an `Unknown value (?)` has no effect.

See also

setDefaultPersistency procedure on page 256, getDefaultPersistency function on page 197, setDefaultPriority procedure on page 256getDefaultPriority function on page 197, setDefaultTimeToLive procedure on page 257, getDefaultTimeToLive function on page 198

For more information, see the Accessing message delivery parameters.

setDefaultTimeToLive procedure

Sets the default time to live, the number of milliseconds from the time a message is sent to the time the JMS-provider Broker can delete the message from the system.

Syntax

```
PROCEDURE setDefaultTimeToLive.
DEFINE INPUT PARAMETER millis AS DECIMAL.
```

Applies to

Session objects

Notes

- A setting of 0 specifies that the message never expires.
- The default is JMS-broker-dependent; the SonicMQ default value is 0.
- Any fractional part of the decimal value is truncated.
- If the value does not fit in a Java `long` value, Java rules for decimal-to-long conversions are used.
- Setting an `Unknown value (?)` has no effect.

See also

setDefaultPersistency procedure on page 256, getDefaultPersistency function on page 197, setDefaultPriority procedure on page 256getDefaultPriority function on page 197, setDefaultTimeToLive procedure on page 257, getDefaultTimeToLive function on page 198

For more information, see the Accessing message delivery parameters.

setDouble procedure

Converts data in a `MapMessage` to the JMS `double` data type.

Syntax

```
PROCEDURE setDouble.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setDoubleProperty procedure

Sets a `double` message property.

Syntax

```
PROCEDURE setDoubleProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

Applies to

Message objects

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

setErrorHandler procedure

Sets the error-handling procedure.

Syntax

```
PROCEDURE setErrorHandler.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

Applies to

Session objects

Notes

- The application must create the error-handling *messageConsumer* object and call setErrorHandler procedure on page 258 after calling beginSession procedure on page 172. If the application does not call setErrorHandler procedure on page 258, a default error handler displays the error message and the properties in alert boxes.
- The application should handle asynchronously reported error conditions programmatically by creating an error-handling Message Consumer object and passing it to setErrorHandler procedure on page 258 in the Session object. setErrorHandler procedure on page 258 creates an error-handling Message Consumer object.
- Asynchronous conditions are always reported as a `TextMessage` with several possible `CHARACTER` message properties. The `CHARACTER` properties that might be included in the message header are: `exception`, `errorCode`, `linkedException-1`, `linkedException-2` … `linkedException-`*n* (where *n* is a number of additional exceptions linked to the main exception).
- getPropertyNames function on page 216 can be used to get the list of properties in the error message header.

See also

setErrorHandler procedure on page 258, setNoErrorDisplay procedure on page 269

For more information, see the Error and condition handling. For an example, see Messaging Examples.

setFloat procedure

Converts data in a `MapMessage` to the JMS `float` data type.

Syntax

```
PROCEDURE setFloat.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setFloatProperty procedure

Sets a `float` message property.

Syntax

```
PROCEDURE setFloatProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

Applies to

Message objects

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

setInt procedure

Converts data in a `MapMessage` to the JMS `int` data type.

Syntax

```
PROCEDURE setInt.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INTEGER.
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt64 procedure on page 261, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getInt64 function on page 200, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setIntProperty procedure

Converts an `int` message property.

Syntax

```
PROCEDURE setIntProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INTEGER.
```

Applies to

Message objects

See also

getIntProperty function on page 199, clearProperties procedure on page 174, setInt64Property procedure on page 262

For more information, see the Accessing message properties.

setInt64 procedure

Converts data in a `MapMessage` to the JMS `long` data type.

Syntax

```
PROCEDURE setInt64.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INT64.
```

Applies to

Message objects

Notes

The `Unknown value (?)` is allowed.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getInt64 function on page 200, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setInt64Property procedure

Sets the value of a JMS message property to `INT64`.

Syntax

```
PROCEDURE setInt64Property.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INT64.
```

Applies to

Message objects

Notes

The `Unknown value (?)` is allowed.

See also

getInt64Property function on page 200, setIntProperty procedure on page 261, getIntProperty function on page 199, clearProperties procedure on page 174

For more information, see the Accessing message properties.

setJMSCorrelationID procedure

Sets the correlation ID.

Syntax

```
PROCEDURE setJMSCorrelationID
DEFINE INPUT PARAMETER correlationID AS CHARACTER.
```

Applies to

Message objects

Notes

This value is application-defined; typically it is set to the ID of the message replied to.

See also

setJMSCorrelationID procedure on page 262, getJMSCorrelationID function on page 201

For more information, see the Accessing message header properties.

setJMSCorrelationIDAsBytes procedure

Sets the bytes correlation ID, a proprietary (JMS-provider-dependent) value.

Syntax

```
PROCEDURE setJMSCorrelationIDAsBytes
DEFINE INPUT PARAMETER bytesCorrelationID AS RAW.
```

Applies to

Message objects

Notes

- The bytes correlation ID usage is proprietary (JMS-provider-dependent).
- When accessing SonicMQ, the *bytesCorrelationID* field can be used for storing application-defined values.

See also

setJMSCorrelationIDAsBytes procedure on page 263, getJMSCorrelationIDAsBytes function on page 202

For more information, see the Accessing message header properties.

setJMSReplyTo procedure

Sets a destination for replies.

Syntax

```
PROCEDURE setJMSReplyTo
DEFINE INPUT PARAMETER destination AS CHARACTER.
```

Applies to

Message objects

Notes

- The destination can be a name of a queue if the message is sent by a Pub/Sub session.
- The destination can be the name of the topic if the message is sent by a PTP session.
- setReplyToDestinationType procedure on page 273 must be called to set the correct destination type.

See also

setJMSReplyTo procedure on page 263, getJMSReplyTo function on page 205, hasReplyTo function on page 227, setReplyToDestinationType procedure on page 273, getReplyToDestinationType function on page 219

For more information, see the Accessing message header properties.

setJMSServerName procedure

Specifies the JMS-provider broker implementation.

Syntax

```
PROCEDURE setJmsServerName.
DEFINE INPUT PARAMETER jmsServerName AS CHARACTER.
```

Parameter

jmsServerName

Specifies the JMS-provider broker implementation. For example, SonicMQ.

Applies to

Session objects

Notes

If set on the client, it overwrites the *jmsServerName* property set on the Progress OpenEdge JMS Adapter side.

See also

setJMSServerName procedure on page 264, getJMSServerName function on page 206

For more information, see the Setting and getting JMS connection and session attributes.

setJMSType procedure

Sets the type name, which is proprietary (JMS-provider-dependent).

Syntax

```
PROCEDURE setJMSType
DEFINE INPUT PARAMETER typeName AS CHARACTER.
```

Applies to

Message objects

Notes

When accessing SonicMQ, the *JMSType* field can be used for storing application-defined values.

See also

setJMSType procedure on page 265, getJMSType function on page 207

For more information, see the Accessing message header properties.

setLoadBalancing procedure

Turns client-side load balancing on or off.

Syntax

```
PROCEDURE setLoadBalancing
DEFINE INPUT PARAMETER loadBalancing AS LOGICAL.
```

Applies to

Session objects

Notes

- If client-side load balancing is turned on, the client allows redirection to another JMS-provider Broker in the cluster.
- If client-side load balancing is turned off, the client does not allow redirection.
- If beginSession procedure on page 172 has already been called, an error is raised.

See also

setLoadBalancing procedure on page 265, getLoadBalancing function on page 207

For more information, see the Load balancing.

setLong procedure

Converts `long` data in a `MapMessage` in Text and XML messages.

Syntax

```
PROCEDURE setLong.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

Applies to

Message objects

Notes

Any fractional part of the `DECIMAL` value is truncated.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setLongProperty procedure

Sets a `long` message property.

Syntax

```
PROCEDURE setLongProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

Applies to

Message objects

Notes

Any fractional part of the `DECIMAL` value is truncated.

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

setLongString procedure

Sets `String` data in a `MapMessage`.

Syntax

```
PROCEDURE setLongString.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMTER value AS LONGCHAR.
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setLongText procedure

Clears the message body and sets a new text or XML value of any length.

Syntax

```
PROCEDURE setLongText.
DEFINE INPUT PARAMTER textValue AS LONGCHAR.
```

Applies to

Message objects

Notes

An error is returned if the `Unknown value (?)` is specified.

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

setMemptr procedure

Sets the specified number of bytes from the `MEMPTR` variable starting at *`startIndex`* in a `BytesMessage`.

Syntax

```
PROCEDURE setMemptr.
DEFINE INPUT PARAMETER memptrVar AS MEMPTR.
DEFINE INPUT PARAMETER startIndex AS INTEGER.
DEFINE INPUT PARAMETER numBytes AS INTEGER.
```

Applies to

Message objects

Notes

- The first byte is 1.
- setMemptr procedure on page 268 implicitly calls clearBody procedure on page 174 before setting the data and resets after setting the data. Therefore, it can be used whether the message is in a read-only mode or a write-only mode prior to the call.
- The call makes a copy of the data. Thus, the *`memptrVar`* variable is not modified by the ABL–JMS implementation and can be modified by the OpenEdge application after the call without corrupting the message.

See also

createBytesMessage procedure on page 176, setMemptr procedure on page 268, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, getContentType on page 148, getBytesCount function on page 187, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239, getMemptr function on page 212

For more information, see the BytesMessage.

For an example, see the Publishing, subscribing, and receiving an XML document in a BytesMessage.

setNoAcknowledge procedure

Instructs the ABL–JMS implementation not to acknowledge this message.

Syntax

```
PROCEDURE setNoAcknowledge.
```

Applies to

Message Consumer objects

Notes

- This call should be made if the OpenEdge application fails to use the data in a message and must receive the message again.
- This call is an error if the session is transacted for receiving.
- If the Message Consumer object is used to handle error messages or for queue browsing, this call has no effect.

See also

acknowledgeAndForward procedure on page 141, setSingleMessageAcknowledgement procedure on page 169, getSingleMessageAcknowledgement function on page 155, setNoAcknowledge procedure on page 269, getNoAcknowledge function on page 214

For more information, see the Message acknowledgement, forwarding, and recovery and the Single-message acknowledgement.

setNoErrorDisplay procedure

Turns the automatic display of synchronous errors and conditions on and off.

Syntax

```
PROCEDURE setNoErrorDisplay.
DEFINE INPUT PARAMETER noDisplay AS LOGICAL.
```

Applies to

Session objects and Message objects

Notes

- The default value is FALSE. The ABL–JMS implementation automatically displays synchronously reported errors and conditions in alert boxes.
- If set to TRUE, synchronous errors and conditions are not automatically displayed by the ABL–JMS implementation.
- Messages inherit the *noDisplay* property from the session that created them.
- After the message is created, setNoErrorDisplay procedure on page 269 must be called in the message itself to change the *noDisplay* property.
- Errors caused by method calls are automatically displayed.

See also

setErrorHandler procedure on page 258, setNoErrorDisplay procedure on page 269

For more information, see the Error and condition handling.

setPassword procedure

Sets the password value for the JMS-provider Broker login and overwrites the default password property set on the Progress OpenEdge JMS Adapter side.

Syntax

```
PROCEDURE setPassword.
DEFINE INPUT PARAMETER password AS CHARACTER.
```

Parameter

password

Password value for the JMS-provider Broker login.

Applies to

Session objects

Notes

If called, setPassword procedure on page 270 overwrites the default password property set on the Progress OpenEdge JMS Adapter side.

See also

getPassword on page 215

For more information, see the Setting and getting JMS connection and session attributes.

setReplyAutoDelete procedure

Specifies whether all reply messages are to be automatically deleted.

Syntax

```
PROCEDURE setReplyAutoDelete.
DEFINE INPUT PARAMETER val AS LOGICAL.
```

Applies to

Message Consumer objects

Notes

- The default value is FALSE.
- If the reply property *val* is set to TRUE, all reply messages returned through the message handler's OUPUT parameter are automatically deleted after being sent.

See also

setReplyPersistency procedure on page 271, getReplyPersistency function on page 218, setReplyPriority procedure on page 272, getReplyPriority function on page 218, setReplyTimeToLive procedure on page 272, getReplyTimeToLive function on page 219, setReplyAutoDelete procedure on page 271, getReplyAutoDelete function on page 217

For more information, see the Accessing message handler information and the Setting reply properties.

setReplyPersistency procedure

Sets the value for message persistency when the Message Consumer is passed to requestReply procedure on page 241.

Syntax

```
PROCEDURE setReplyPersistency.
DEFINE INPUT PARAMETER deliveryMode AS CHARACTER.
```

Applies to

Message Consumer objects

Notes

- The values are: PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, and UNKNOWN. The default value is PERSISTENT.
- The evaluation is case-insensitive. A call with an Unknown value (?) has no effect.

- The `replyPersistency` value can be set only once.
- `NON_PERSISTENT_ASYNC` is a SonicMQ extension.

See also

setReplyPersistency procedure on page 271, getReplyPersistency function on page 218, setReplyPriority procedure on page 272, getReplyPriority function on page 218, setReplyTimeToLive procedure on page 272, getReplyTimeToLive function on page 219, setReplyAutoDelete procedure on page 271, getReplyAutoDelete function on page 217

For more information, see the Accessing message handler information and the Setting reply properties.

setReplyPriority procedure

Sets the priority of the reply messages when the Message Consumer is passed to requestReply procedure on page 241.

Syntax

```
PROCEDURE setReplyPriority.
DEFINE INPUT PARAMETER priority AS INTEGER.
```

Applies to

Message Consumer objects

Notes

- The range of values is 0–9; the default is 4.
- This procedure can be called only once.

See also

setReplyPersistency procedure on page 271, getReplyPersistency function on page 218, setReplyPriority procedure on page 272, getReplyPriority function on page 218, setReplyTimeToLive procedure on page 272, getReplyTimeToLive function on page 219, setReplyAutoDelete procedure on page 271, getReplyAutoDelete function on page 217

For more information, see the Accessing message handler information and the Setting reply properties.

setReplyTimeToLive procedure

Sets the time to live value (in milliseconds) of the reply messages when the Message Consumer is passed to requestReply procedure on page 241.

Syntax

```
PROCEDURE setReplyTimeToLive.
DEFINE INPUT PARAMETER millis AS DECIMAL.
```

Applies to

Message Consumer objects

Notes

- Time to live is the number of milliseconds from the time the message is sent to the time the JMS-provider Broker can delete the message from the system.
- A value of 0 means the message never expires.
- The default is JMS-system-dependent; the SonicMQ default value is 0.
- The `replyTimeToLive` values can be set only once. The fractional part of the decimal value is truncated. If the value does not fit in a Java `long` value, Java rules for decimal-to-long conversion apply.

See also

setReplyPersistency procedure on page 271, getReplyPersistency function on page 218, setReplyPriority procedure on page 272, getReplyPriority function on page 218, setReplyTimeToLive procedure on page 272, getReplyTimeToLive function on page 219, setReplyAutoDelete procedure on page 271, getReplyAutoDelete function on page 217

For more information, see the Accessing message handler information and the Setting reply properties.

setReplyToDestinationType procedure

Sets the type of the destination specified by setJMSReplyTo procedure on page 263.

Syntax

```
PROCEDURE setReplyToDestinationType
DEFINE INPUT PARAMETER type AS CHARACTER.
```

Applies to

Message objects

Notes

- The *type* value can be `queue` or `topic`.
- If setReplyToDestinationType procedure on page 273 is not called, a default type is automatically set when the message is sent, according to the type of the session: `queue` for PTP or `topic` for Pub/Sub.
- For a `jmsSession`, the default is queue. To use topics, call setReplyToDestinationType procedure on page 273 to set `topic`.

See also

setJMSReplyTo procedure on page 263, getJMSReplyTo function on page 205, hasReplyTo function on page 227, setReplyToDestinationType procedure on page 273, getReplyToDestinationType function on page 219

For more information, see the Accessing message header properties.

setReuseMessage procedure

Instructs the Message Consumer object not to create a new message for each received message.

Syntax

```
PROCEDURE setReuseMessage.
```

Applies to

Message Consumer objects

Notes

- Calling setReuseMessage procedure on page 274 improves performance. If the procedure is not called, the Message Consumer object creates a new message for each received message.
- A message that is being reused should not be deleted before the session is deleted.

See also

setReuseMessage procedure on page 274, getReuseMessage function on page 220

For more information, see the Reusing messages.

setSaxReader procedure

Sets the input destination to an internal longchar that will be used as intermediate storage of the XML read from an `XMLMessage`.

Syntax

```
PROCEDURE setSaxReader.
INPUT PARAMETER hdl AS HANDLE.
```

Applies to

Message objects

Notes

- The specified handle must already be initialized as a SAX-READER handle. If the specified handle is not a SAX-READER handle, an error is returned.
- The caller uses SAX-READER methods on the handle to read the XML document.

See also

createXMLMessage procedure on page 181, setX-Document procedure on page 280, getX-Document function on page 226, setSaxReader procedure on page 274, getSaxWriter function on page 221, deleteSaxWriter procedure on page 183

For more information, see the XMLMessage.

setShort procedure

Converts data to the JMS `short` data type in a `MapMessage`.

Syntax

```
PROCEDURE setShort.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS INTEGER.
```

Applies to

Message objects

Notes

The server returns a NumberFormatException message for a value overflow.

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setShortProperty procedure

Sets a `short` message property.

Syntax

```
PROCEDURE setShortProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS INTEGER.
```

Applies to

Message objects

Notes

The server returns a NumberFormatException message for a value overflow.

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

setShutdownWaitFor procedure

Sets a LOGICAL value indicating the setting of the shutdown WAIT-FOR flag.

Syntax

```
PROCEDURE setShutdownWaitFor.
DEFINE INPUT PARAMETER val AS LOGICAL.
```

Applies to

Session objects

Notes

- The default behavior of deleteSession procedure on page 183 is to execute the ABL statement WAIT-FOR PROCEDURE COMPLETE. The WAIT-FOR statement in deleteSession procedure on page 183 can cause an error if your application has an outstanding WAIT-FOR call. You can direct deleteSession procedure on page 183 to skip the WAIT-FOR call, by calling setShutdownWaitFor procedure on page 276 with a logical "NO" input parameter.
- If you direct deleteSession procedure on page 183 to skip the WAIT-FOR call, your application must issue a blocking-I/O statement to allow the adapter to process the shutdown message.
- The default value for the shutdown WAIT-FOR flag is YES, which tells deleteSession procedure on page 183 to execute the ABL statement WAIT-FOR PROCEDURE COMPLETE.

See also

deleteSession procedure on page 183, getShutdownWaitFor function on page 222.

setString procedure

Converts data in a `MapMessage` to the JMS `String` data type.

Syntax

```
PROCEDURE setString.
DEFINE INPUT PARAMETER itemName AS CHARACTER.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

Applies to

Message objects

See also

createMapMessage procedure on page 177, setBoolean procedure on page 246, setByte procedure on page 248, setBytesFromRaw procedure on page 249, setChar procedure on page 250, setDate procedure on page 251, setDateTime procedure on page 253, setDateTime-TZ procedure on page 254, setDouble procedure on page 257, setFloat procedure on page 259, setInt procedure on page 260, setLong procedure on page 266, setLongString procedure on page 267, setShort procedure on page 275, setString procedure on page 277, getMapNames function on page 211, getItemType function on page 201, getBytesToRaw function on page 188, getChar function on page 188, getDate function on page 192, getDateTime function on page 193, getDateTime-TZ function on page 195, getDecimal function on page 196, getInt function on page 199, getLogical function on page 208, getLongString function on page 209, getLongStringCP function on page 209

For more information, see the MapMessage.

setStringProperty procedure

Sets a `String` message property.

Syntax

```
PROCEDURE setStringProperty.
DEFINE INPUT PARAMETER propertyName AS CHARACTER.
DEFINE INPUT PARAMETER propertyValue AS CHARACTER.
```

Applies to

Message objects

See also

clearProperties procedure on page 174

For more information, see the Accessing message properties.

setTempTable procedure

Sets the TempTableMessage.

Syntax

```
PROCEDURE setTempTable.
DEFINE INPUT PARAMETER tableHdl AS HANDLE.
DEFINE INPUT PARAMETER schemaLocation AS CHARACTER.
DEFINE INPUT PARAMETER writeSchema AS LOGICAL.
```

Applies to

Message objects

Notes

- The handle parameter must be a handle to a valid TempTable.
- The schema parameters specify the schema information and are passed directly to the WRITE-XML method. Specifying an Unknown value (?) for any of the schema parameters will result in the use of the default value for that parameter.
- The TempTableMessage is converted to an XMLMessage using the WRITE-XML method and setLongText procedure on page 267.

See also

createTempTableMessage procedure on page 180, setTempTable procedure on page 278, getTempTable function on page 222

For more information, see the TempTableMessage.

setText procedure

Clears the message body and sets a new text value.

Syntax

```
PROCEDURE setText.
DEFINE INPUT PARAMETER textValue AS CHARACTER.
```

Applies to

Message objects

Notes

- The call can be made when the message is in write-only or read-only mode.

- After the call, the message is in write-only mode. You can use appendText procedure on page 171 calls to append more text.
- As an alternative to concatenating multiple CHARACTER segments with the appendText procedure on page 171, use setLongText procedure on page 267.

See also

createTextMessage procedure on page 181, setText procedure on page 278, setLongText procedure on page 267, appendText procedure on page 171, endOfStream function on page 185, getCharCount function on page 189, getText function on page 223, getTextSegment function on page 224, getLongText function on page 210, getLongTextCP function on page 210

For more information, see the TextMessage.

setTransactedReceive procedure

Makes the session transacted for receiving.

Syntax

```
PROCEDURE setTransactedReceive.
```

Applies to

Session objects

Notes

A session is not transacted by default.

See also

setTransactedReceive procedure on page 279, getTransactedReceive function on page 225, setTransactedSend procedure on page 279, getTransactedSend function on page 225

For more information, see the Setting and getting JMS connection and session attributes.

setTransactedSend procedure

Makes the session transacted for sending.

Syntax

```
PROCEDURE setTransactedSend.
```

Applies to

Session objects

Notes

A session is not transacted by default.

See also

setTransactedReceive procedure on page 279, getTransactedReceive function on page 225, setTransactedSend procedure on page 279, getTransactedSend function on page 225

For more information, see the Setting and getting JMS connection and session attributes.

setUser procedure

Sets the *user* value for the JMS-provider Broker login and overwrites the default *user* property set on the Progress OpenEdge JMS Adapter side.

Syntax

```
PROCEDURE setUser.
DEFINE INPUT PARAMETER user AS CHARACTER.
```

Parameter

user

JMS-provider Broker login.

Applies to

Session objects

Notes

If called, setUser procedure on page 280 overwrites the default user property set on the Progress OpenEdge JMS Adapter side.

See also

setUser procedure on page 280, getUser function on page 226

For more information, see the Setting and getting JMS connection and session attributes.

setX-Document procedure

Copies an XML document specified by the handle parameter into the `XMLMessage`.

Syntax

```
PROCEDURE setX-Document.
DEFINE INPUT PARAMETER hdl AS HANDLE.
```

Applies to

Message objects

Notes

- The caller must have properly defined the handle and correctly loaded an XML document using X-DOCUMENT calls. If the handle is not an X-DOCUMENT or X-NODOREF handle, an error is returned.
- This procedure loads the XML into an internal longchar. Use setLongText procedure on page 267 to copy the longchar into the XMLMessage.
- Other errors include any error generated by xdoc:SAVE, any error generated by the XMLMessage, or by setLongText procedure on page 267.

See also

createXMLMessage procedure on page 181, setX-Document procedure on page 280, getX-Document function on page 226, setSaxReader procedure on page 274, getSaxWriter function on page 221, deleteSaxWriter procedure on page 183

For more information, see the XMLMessage.

startReceiveMessages procedure

Starts receiving messages after creating a new session or after calling stopReceiveMessages procedure on page 282.

Syntax

```
PROCEDURE startReceiveMessages.
```

Applies to

Session objects

Notes

- Messages can be sent without calling startReceiveMessages procedure on page 281.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

startReceiveMessages procedure on page 281, stopReceiveMessages procedure on page 282

For more information, see the Establishing session control.

stopReceiveMessages procedure

Causes the Progress OpenEdge JMS Adapter Broker to stop receiving messages on behalf of the OpenEdge client.

Syntax

```
PROCEDURE stopReceiveMessages.
```

Applies to

Session objects

Notes

- A subsequent call to startReceiveMessages procedure on page 281 resumes message reception and delivery.
- If this procedure is called in a `pubsubsession` object and the subscription is not durable, messages published while reception is stopped are not delivered.
- A single message that was already sent to the client before stopReceiveMessages procedure on page 282 was called might be received by the client after stopReceiveMessages procedure on page 282 call.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

startReceiveMessages procedure on page 281, stopReceiveMessages procedure on page 282

For more information, see the Establishing session control.

subscribe procedure

Subscribes to a topic.

Syntax

```
PROCEDURE subscribe.
DEFINE INPUT PARAMETER topicName AS CHARACTER.
DEFINE INPUT PARAMETER subscriptionName AS CHARACTER.
DEFINE INPUT PARAMETER messageSelector AS CHARACTER.
DEFINE INPUT PARAMETER noLocalPublications AS LOGICAL.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

Parameters

topicName

The topic to which the procedure subscribes.

subscriptionName

A *subscriptionName* parameter with a value other than UNKNOWN specifies a durable subscription. Durable subscriptions require that the JMS client have a client ID identifier; the client must call setClientID in the pubsubsession.p object (or set the default client ID on the server side). If the *subscriptionName* value is UNKNOWN, the subscription is not durable.

messageSelector

A message selector (optional).

noLocalPublications

A Boolean flag controlling whether the application receives its own messages (optional). The default is FALSE (the session receives its own publications).

messageConsumer

The Message Consumer object.

Applies to

Session objects

Notes

- The messages are handled asynchronously by the *messageConsumer* object.
- If the *subscriptionName* value is UNKNOWN or an empty string, the subscription is not durable.
- Durable subscriptions require the JMS client to have a client ID identifier.
- The client must call setClientID procedure on page 250 in the pubsubsession object (or set the default client ID on the server side) if a durable subscription is desired. The default of *noLocalPublications* is FALSE. The session, by default, get its own publications.
- This procedure executes remotely on the Progress OpenEdge JMS Adapter.

See also

cancelDurableSubscription procedure on page 173, publish procedure on page 232, subscribe procedure on page 282

For more information, see the Publishing messages to a topic, the Subscribing to a topic, the Durable subscriptions, and the Methods unique to Pub/Sub messaging.

For an example, see the Pub/Sub messaging example on page 85.

waitForMessages procedure

Waits and processes events as long as the user-defined function is TRUE.

Syntax

```
PROCEDURE waitForMessages:
DEFINE INPUT PARAMETER UDFName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER procH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER timeOut AS INTEGER NO-UNDO.
```

Parameters

UDFName

User-defined function.

procH

Handle to user-defined function.

timeOut

Time limit (in seconds) in which no messages are received.

Applies to

Session objects

Notes

- *UDFName* (in *procH*) returns TRUE and there is no period of more than *timeOut* seconds in which no messages are received.
- The user-defined function, *UDFName*, is evaluated each time after a message is handled.

See also

createMessageConsumer procedure on page 178, deleteConsumer procedure on page 182, messageHandler procedure on page 230, waitForMessages procedure on page 283

For more information see the Message Consumer objects, the Consuming messages, the Terminating the Message Consumer object, and the Creating a message handler process.

writeBoolean procedure

Writes `boolean` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeBoolean.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

Applies to

Message objects

Notes

An `Unknown value (?)` is considered FALSE.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeByte procedure

Writes `byte` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeByte.
DEFINE INPUT PARAMETER value AS INTEGER.
```

Applies to

Message objects

Notes

- Byte values range from –128 to 127.
- The server returns a NumberFormatException message for a value overflow. For example, calling `writeByte(1000)` results in a value overflow.
- Used in write-only mode to write an additional bytes segment to a `BytesMessage`.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeBytesFromRaw procedure

Writes `byte` array data to the body of a `StreamMessage` or `ByteMessage`.

Syntax

```
PROCEDURE writeBytesFromRaw.
DEFINE INPUT PARAMETER bytesValue AS RAW.
```

Applies to

Message objects

Notes

- This procedure applies to both `StreamMessage` and `ByteMessage` message types.
- This procedure can be called in write-only mode to write additional bytes segments to a `BytesMessage` and work around the `RAW` data type limit of 32K.

See also

createStreamMessage procedure on page 179, getContentType on page 148, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage and the BytesMessage.

writeChar procedure

Writes `char` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeChar.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

Applies to

Message objects

Notes

The number of characters in the `char` value must be one.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeDate procedure

Writes a `date` value as `String` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeDate.
DEFINE INPUT PARAMETER value AS DATE.
```

Applies to

Message objects

Notes

- The procedure actually writes *value* as a `DATETIME-TZ` value, adding default time and time zone information:
  - A time of 12:00AM (midnight)
  - The default time zone of the client application
- If the message might be consumed by an application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly.
- The format of the `DATETIME-TZ` value is a subset of the ISO8601 format.
- An error is returned if the `Unknown value (?)` is specified.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeDateTime procedure

Writes a `date-time` value as `String` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeDateTime.
DEFINE INPUT PARAMETER value AS DATETIME.
```

Applies to

Message objects

Notes

- The procedure actually writes *value* as a `DATETIME-TZ` value, adding default time and time zone information.
- The format of the `DATETIME-TZ` value is a subset of the ISO8601 format.
- An error is returned if the `Unknown value (?)` is specified.
- If the message may be consumed by an application in a different time zone from that of the sending application, the ABL programmer must ensure that the receiving application can interpret the value correctly.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeDateTime-TZ procedure

Writes a date-time value, including time zone information, as `String` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeDateTime-TZ.
DEFINE INPUT PARAMETER value AS DATETIME-TZ.
```

Applies to

Message objects

Notes

An error is returned if the `Unknown value (?)` is specified.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeDouble procedure

Writes `double` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeDouble.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeFloat procedure

Writes `float` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeFloat.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

Applies to

Message objects

Notes

The fractional part of the `DECIMAL` value is truncated.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeInt procedure

Writes `int` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeInt.
DEFINE INPUT PARAMETER value AS INTEGER.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt64 procedure on page 291, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readInt64 function on page 238, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeInt64 procedure

Writes `INT64` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeInt64.
DEFINE INPUT PARAMETER value AS INT64.
```

Applies to

Message objects

Notes

The `Unknown value (?)` is allowed.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readInt64 function on page 238, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeLong procedure

Writes `long` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeLong.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

Applies to

Message objects

Notes

The fractional part of the `DECIMAL` value is truncated.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeLongString procedure

Writes `String` data of any length to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeLongString.
DEFINE INPUT PARAMETER value AS LONGCHAR.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeShort procedure

Writes `short` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeShort.
DEFINE INPUT PARAMETER value AS INTEGER.
```

Applies to

Message objects

Notes

The server returns a NumberFormatException message for a value overflow.

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.

writeString procedure

Writes `String` data to the body of a `StreamMessage`.

Syntax

```
PROCEDURE writeString.
DEFINE INPUT PARAMETER value AS CHARACTER.
```

Applies to

Message objects

See also

createStreamMessage procedure on page 179, writeBoolean procedure on page 284, writeByte procedure on page 285, writeBytesFromRaw procedure on page 286, writeChar procedure on page 286, writeDate procedure on page 287, writeDateTime procedure on page 288, writeDateTime-TZ procedure on page 289, writeDouble procedure on page 289, writeFloat procedure on page 290, writeInt procedure on page 290, writeLong procedure on page 292, writeLongString procedure on page 292, writeShort procedure on page 293, writeString procedure on page 294, endOfStream function on page 185, moveToNext procedure on page 231, readBytesToRaw procedure on page 233, readChar function on page 234, readDate function on page 234, readDateTime function on page 235, readDateTime-TZ function on page 236, readDecimal function on page 236, readInt function on page 237, readLogical function on page 238, readLongString function on page 239, readLongStringCP function on page 239

For more information, see the StreamMessage.