

# **TIBCO Enterprise Message Service™**

## **.NET Reference**

*System Release 4.3  
February 2006*

## Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE *TIBCO ENTERPRISE MESSAGE SERVICE USER'S GUIDE*). USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIB, TIBCO, Information Bus, The Power of Now, TIBCO Adapter, Rendezvous are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

EJB, J2EE, JMS and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

Copyright © 1999–2006 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information

# Contents

<b>Tables</b>	<b>ix</b>
<b>Preface</b>	<b>xi</b>
Related Documentation	xii
TIBCO Enterprise Message Service Documentation	xii
Other TIBCO Product Documentation	xii
Third Party Documentation	xii
How to Contact TIBCO Customer Support	xiv
<b>Chapter 1 Introduction</b>	<b>1</b>
Overview	2
Excluded Features and Restrictions	3
Object Serialization	3
Strings and Character Encodings	4
.NET Compact Framework (CF)	5
<b>Chapter 2 Programmer's Checklist</b>	<b>7</b>
Install	7
Code	7
Compile	7
Run	7
<b>Chapter 3 Messages</b>	<b>9</b>
Parts of a Message	10
Body Types	11
Headers	12
Properties	17
Setting Message Properties	17
EMS Properties	17
JMS Properties	19
Message Selectors	20
Data Type Conversion	23
Message	24
Message.Acknowledge	27

Message.ClearBody . . . . .	28
Message.ClearProperties . . . . .	29
Message.Clone . . . . .	30
Message.GetDeliveryModeName . . . . .	31
Message—Get Properties . . . . .	32
Message.PropertyExists . . . . .	33
Message—Set Properties . . . . .	34
Message.ToString . . . . .	35
BytesMessage . . . . .	36
BytesMessage—Read . . . . .	37
BytesMessage.ReadBytes . . . . .	39
BytesMessage—Write . . . . .	40
BytesMessage.WriteBytes . . . . .	42
BytesMessage.Reset . . . . .	43
MapMessage . . . . .	44
MapMessage—Get . . . . .	46
MapMessage.ItemExists . . . . .	47
MapMessage—Set . . . . .	48
MapMessage.SetBytes . . . . .	49
ObjectMessage . . . . .	50
ObjectMessage . . . . .	51
StreamMessage . . . . .	52
StreamMessage—Read . . . . .	54
StreamMessage.ReadBytes . . . . .	55
StreamMessage.Reset . . . . .	56
StreamMessage—Write . . . . .	57
StreamMessage.WriteBytes . . . . .	58
TextMessage . . . . .	59
TextMessage . . . . .	60
<b>Chapter 4 Destination . . . . .</b>	<b>61</b>
Destination Overview . . . . .	62
Destination . . . . .	65
Queue . . . . .	66
Queue . . . . .	67
TemporaryQueue . . . . .	68
TemporaryQueue.Delete . . . . .	69
TemporaryTopic . . . . .	70
TemporaryTopic.Delete . . . . .	71
Topic . . . . .	72
Topic . . . . .	73

<b>Chapter 5 Consumer</b>	<b>75</b>
MessageConsumer	76
MessageConsumer.Close	78
MessageConsumer.Receive	79
MessageConsumer.ReceiveNoWait	80
QueueReceiver	81
TopicSubscriber	82
EMSMessageHandler	83
EMSMessageEventArgs	84
EMSMessageEventArgs	85
IMessageListener	86
IMessageListener.OnMessage	87
<b>Chapter 6 Producer</b>	<b>89</b>
MessageProducer	90
MessageProducer.Close	93
MessageProducer.Send	94
QueueSender	96
QueueSender.Send	97
TopicPublisher	99
TopicPublisher.Publish	100
<b>Chapter 7 Requestor</b>	<b>103</b>
QueueRequestor	104
QueueRequestor	105
QueueRequestor.Close	106
QueueRequestor.Request	107
TopicRequestor	108
TopicRequestor	109
TopicRequestor.Close	110
TopicRequestor.Request	111
<b>Chapter 8 Connection</b>	<b>113</b>
Connection	114
Connection.Close	117
Connection.CreateSession	118
Connection.Start	119
Connection.Stop	120
ConnectionMetaData	121
QueueConnection	122

QueueConnection.CreateQueueSession . . . . .	123
TopicConnection . . . . .	124
TopicConnection.CreateTopicSession . . . . .	125
EMSEExceptionHandler . . . . .	126
EMSEExceptionEventArgs . . . . .	127
EMSEExceptionEventArgs . . . . .	128
IExceptionListener . . . . .	129
IExceptionListener.OnException . . . . .	130
<b>Chapter 9 Connection Factory . . . . .</b>	<b>131</b>
ConnectionFactory . . . . .	132
ConnectionFactory . . . . .	134
ConnectionFactory.CreateConnection . . . . .	135
FactoryLoadBalanceMetric . . . . .	136
QueueConnectionFactory . . . . .	137
QueueConnectionFactory.CreateQueueConnection . . . . .	138
TopicConnectionFactory . . . . .	139
TopicConnectionFactory.CreateTopicConnection . . . . .	140
<b>Chapter 10 Session . . . . .</b>	<b>141</b>
Session . . . . .	142
Session.Close . . . . .	148
Session.Commit . . . . .	149
Session.CreateBrowser . . . . .	150
Session.CreateBytesMessage . . . . .	151
Session.CreateConsumer . . . . .	152
Session.CreateDurableSubscriber . . . . .	153
Session.CreateMapMessage . . . . .	155
Session.CreateObjectMessage . . . . .	156
Session.CreateProducer . . . . .	157
Session.CreateQueue . . . . .	158
Session.CreateStreamMessage . . . . .	159
Session.CreateTemporaryQueue . . . . .	160
Session.CreateTemporaryTopic . . . . .	161
Session.CreateTextMessage . . . . .	162
Session.CreateTopic . . . . .	163
Session.Recover . . . . .	164
Session.Rollback . . . . .	165
Session.Run . . . . .	166
Session.Unsubscribe . . . . .	167
SessionMode . . . . .	168

QueueSession .....	170
TopicSession .....	171
<b>Chapter 11 Queue Browser .....</b>	<b>173</b>
QueueBrowser .....	174
QueueBrowser.Close .....	176
QueueBrowser.GetEnumerator .....	177
QueueBrowser.MoveNext .....	178
QueueBrowser.Reset .....	179
<b>Chapter 12 Name Server Lookup .....</b>	<b>181</b>
LookupContext .....	182
LookupContext .....	184
LookupContext.AddSettings .....	185
LookupContext.Lookup .....	186
LookupContext.RemoveSettings .....	187
<b>Chapter 13 Utilities .....</b>	<b>189</b>
DeliveryMode .....	190
IEMSSerializable .....	191
IEMSSerializable.Deserialize .....	192
IEMSSerializable.Serialize .....	193
MessageDeliveryMode .....	194
Tibems .....	195
Tibems.CalculateMessageSize .....	200
Tibems.CreateFromBytes .....	201
Tibems.GetAllowCloseInCallback .....	202
Tibems.GetAsBytes .....	203
Tibems.GetConnectAttempts .....	204
Tibems.GetEncoding .....	205
Tibems.GetExceptionOnFTSwitch .....	206
Tibems.GetMessageEncoding .....	207
Tibems.GetMessageSize .....	208
Tibems.GetPingInterval .....	209
Tibems.GetProperty .....	210
Tibems.GetReconnectAttempts .....	212
Tibems.GetSessionDispatcherDaemon .....	213
Tibems.GetSocketReceiveBufferSize .....	214
Tibems.GetSocketSendBufferSize .....	215
Tibems.MakeWriteable .....	216
Tibems.SetAllowCloseInCallback .....	217
Tibems.SetConnectAttempts .....	218

Tibems.SetEncoding . . . . .	219
Tibems.SetExceptionOnFTSwitch . . . . .	220
Tibems.SetMessageEncoding . . . . .	221
Tibems.SetPingInterval . . . . .	222
Tibems.SetProperty . . . . .	223
Tibems.SetReconnectAttempts . . . . .	225
Tibems.SetSessionDispatcherDaemon . . . . .	226
Tibems.SetSocketReceiveBufferSize . . . . .	227
Tibems.SetSocketSendBufferSize . . . . .	228
<b>Chapter 14 Exception . . . . .</b>	<b>229</b>
EMSEException . . . . .	230
AuthenticationException . . . . .	232
CannotProceedException . . . . .	233
CommunicationException . . . . .	234
ConfigurationException . . . . .	235
IllegalStateException . . . . .	236
InvalidClientIDException . . . . .	237
InvalidDestinationException . . . . .	238
InvalidNameException . . . . .	239
InvalidSelectorException . . . . .	240
MessageEOFException . . . . .	241
MessageFormatException . . . . .	242
MessageNotReadableException . . . . .	243
MessageNotWriteableException . . . . .	244
NameNotFoundException . . . . .	245
NamingException . . . . .	246
ResourceAllocationException . . . . .	247
SecurityException . . . . .	248
ServiceUnavailableException . . . . .	249
TransactionInProgressException . . . . .	250
TransactionRolledBackException . . . . .	251
<b>Index . . . . .</b>	<b>253</b>



# Tables

Table 1	Feature Support . . . . .	3
Table 2	EMS Assembly DLL . . . . .	7
Table 3	Message Header Names . . . . .	12
Table 4	Message Property Names . . . . .	17
Table 5	Data Type Conversion . . . . .	23
Table 6	BytesMessage Read Methods . . . . .	37
Table 7	BytesMessage Write Methods . . . . .	40
Table 8	Destination Overview . . . . .	62



# Preface

TIBCO Enterprise Message Service™ software lets application programs send and receive messages according to the Java Message Service (JMS) protocol. It also integrates with TIBCO Rendezvous and TIBCO SmartSockets message products.



**This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.**

## Topics

---

- *Related Documentation, page xii*
- *How to Contact TIBCO Customer Support, page xiv*

## Related Documentation

---

This section lists documentation resources you may find useful.

### TIBCO Enterprise Message Service Documentation

The following documents form the TIBCO Enterprise Message Service documentation set:

- *TIBCO Enterprise Message Service User's Guide* Read this manual to gain an overall understanding of the product, its features, and configuration.
- *TIBCO Enterprise Message Service Installation* Read the relevant sections of this manual before installing this product.
- *TIBCO Enterprise Message Service Application Integration Guide* This manual presents detailed instructions for integrating TIBCO Enterprise Message Service with third-party products.
- *TIBCO Enterprise Message Service C & COBOL API Reference* This reference is available in HTML and PDF formats.
- *TIBCO Enterprise Message Service Java API Reference* This reference is available as JavaDoc, and you can access the reference only through the HTML documentation interface.
- *TIBCO Enterprise Message Service .NET API Reference* This reference is available in PDF and HTML format.
- *TIBCO Enterprise Message Service Release Notes* Release notes summarize new features, changes in functionality, and closed issues. This document is available only in PDF format.

### Other TIBCO Product Documentation

You may find it useful to read the documentation for the following TIBCO products:

- TIBCO Rendezvous™ software
- TIBCO SmartSockets™ software

### Third Party Documentation

- Java™ Message Service specification, available through [java.sun.com/products/jms/index.html](http://java.sun.com/products/jms/index.html)

- *Java™ Message Service* by Richard Monson-Haefel and David A. Chappell, O'Reilly and Associates, Sebastopol, California, 2001.

## How to Contact TIBCO Customer Support

---

For comments or problems with this manual or the software it addresses, please contact TIBCO Support Services as follows.

- For an overview of TIBCO Support Services, and information about getting started with TIBCO Product Support, visit this site:

<http://www.tibco.com/services/support/default.jsp>

- If you already have a valid maintenance or support contract, visit this site:

<http://support.tibco.com>

Entry to this site requires a username and password. If you do not have a username, you can request one.

## Chapter 1 Introduction

This chapter presents concepts specific to the TIBCO Enterprise Message Service™ .NET API and .NET Compact Framework API. For more general information and concepts pertaining to TIBCO Enterprise Message Service (EMS) software, see the book *TIBCO Enterprise Message Service User's Guide*.

### Topics

---

- *Overview, page 2*
- *Excluded Features and Restrictions, page 3*
- *Strings and Character Encodings, page 4*

## Overview

---

TIBCO Enterprise Message Service .NET API implements (and extends) the JMS 1.1 specification. It is compatible with the JMS 1.0.2 specification.

This implementation consists of fully-managed .NET code.

TIBCO Enterprise Message Service .NET API closely mimics the Java API. This parallelism eases porting of programs between the two programming languages.

The .NET API incorporates .NET-style event handling and enumerated constants (to enable compiler type checking). These features encourage programmers to use a .NET idiom, even while their Java-oriented versions remain available for quick porting.

EMS .NET Compact Framework API brings the power of EMS to any hand-held device or embedded system that supports .NET Compact Framework.



## Excluded Features and Restrictions

This section summarizes features that are not available in either the .NET library, or the .NET Compact Framework library.

Table 1 Feature Support

Feature	.NET	.NET Compact Framework
XA protocols for external transaction managers	—	—
ConnectionConsumer, ServerSession, ServerSessionPool	—	—
Compression	—	—
SSL	—	—
Modify socket buffer sizes (see <code>Tibems.SetSocketReceiveBufferSize</code> on page 227 and <code>Tibems.SetSocketSendBufferSize</code> on page 228).	Yes	—
Daemon threads (see <code>Tibems.SetSessionDispatcherDaemon</code> on page 226).	Yes	—

## Object Serialization

The .NET library supports serialization for all objects. In contrast, the .NET Compact Framework library supports serialization for a restricted set of objects. For details, see `ObjectMessage` on page 50.

Object serialization differs among the various EMS language APIs in ways that are incompatible. An `ObjectMessage` contains a serialized object. Therefore EMS programs can only send an `ObjectMessage` to another program written in the same language; for example, Java to Java, C to C, .NET to .NET, and .NET Compact Framework to .NET Compact Framework. In particular, notice that a .NET Compact Framework client and a full .NET client cannot exchange an `ObjectMessage`.

## Strings and Character Encodings

---

.NET programs represent strings within messages as byte arrays. Before sending an outbound message, EMS programs translate strings to their byte representation using an encoding, which the program specifies. Conversely, when EMS programs receive inbound messages, they reconstruct strings from byte arrays using the same encoding.

When a program specifies an encoding, it applies to all strings in message bodies (names and values), and properties (names and values). It does *not* apply to header names nor values. The method `BytesMessage.WriteUTF` always uses UTF-8 as its encoding.

For a list of standard encoding names, see [www.iana.org](http://www.iana.org).

Outbound Messages	<p>Programs can determine the encoding of strings in outbound messages in three ways:</p> <ul style="list-style-type: none"><li>• Use the default global encoding—namely, UTF-8.</li><li>• Set a non-default global encoding (for all outbound messages) using <code>Tibems.SetEncoding</code> on page 219.</li><li>• Set the encoding for an individual message using <code>Tibems.SetMessageEncoding</code> on page 221.</li></ul>
Outbound Messages	<p>An inbound message from another EMS client explicitly announces its encoding. A receiving client decodes the message using the proper encoding.</p>

## .NET Compact Framework (CF)

---

This section presents recommendations for using the EMS .NET Compact Framework API to develop applications for handheld devices.

Threads	.NET Compact Framework does not support background threads. To avoid problems with threads, we recommend that programs release all EMS resources before terminating. For example, close EMS connections when they are no longer needed (see <code>Connection.Close</code> on page 117).
Clock Resolution	Clock resolution affects the granularity of all time-related calls and parameters—for example <code>MessageConsumer.Receive(timeout)</code> , connect delays. On some handheld devices, clock resolution is coarser than one might expect. Check the resolution on your target device before selecting time values.
Object Serialization	See Object Serialization on page 3.
Excluded Features	See Excluded Features and Restrictions on page 3.
DLL	See Table 2 on page 7.



## Chapter 2 Programmer's Checklist

Developers of EMS programs can use this checklist during the four phases of the development cycle.

### Install

- Install the EMS software release, which automatically includes the EMS assembly DLLs in the `clients\cs` subdirectory.

### Code

- Import the correct EMS assembly (see Table 2).

Table 2 EMS Assembly DLL

Version	DLL
Full .NET API	TIBCO.EMS.dll
.NET Compact Framework API	TIBCO.EMS-CF.dll

### Compile

- Compile with any .NET compiler.

### Run

- The EMS assembly must be in the global assembly cache (this location is preferred), or in the system path, or in the same directory as your program executable.
- The application must be able to connect to a EMS daemon process (`tibemsd`).



## Chapter 3      **Messages**

Message objects carry application data between client program processes. This chapter presents the structure of messages, JMS message selector syntax to specify a subset of messages based on their property values, the message classes and their methods.

### Topics

---

- *Parts of a Message, page 10*
- *Body Types, page 11*
- *Headers, page 12*
- *Properties, page 17*
- *Message Selectors, page 20*
- *Message, page 24*
- *BytesMessage, page 36*
- *MapMessage, page 44*
- *ObjectMessage, page 50*
- *StreamMessage, page 52*
- *TextMessage, page 59*

## Parts of a Message

---

Messages consist of three parts:

- **Body** The body of a message bears the information content of an application. Several types of message body organize that information in different ways; see Body Types on page 11.
- **Header** Headers associate a fixed set of header field names with values. Clients and providers use headers to identify and route messages.
- **Properties** Properties associate an extensible set of property names with values. The EMS server uses properties to attach ancillary information to messages. Client applications can also use properties—for example, to customize message filtering.



## Body Types

---

EMS follows JMS in defining five types of message body:

- **MapMessage** The message body is a mapping from field names to values. Field names are strings. EMS supports an extended set of values types (extending JMS). Programs can access fields either by name, or sequentially (though the order of that sequence is indeterminate).
- **ObjectMessage** The message body is one serializable object.
- **StreamMessage** The message body is a stream of values. Programs write the values sequentially into the stream, and read values sequentially from the stream.
- **TextMessage** The message body is one character string (of any length). This text string can represent any text, including an XML document.
- **BytesMessage** The message body is a stream of uninterpreted bytes. Programs can use this body type to emulate body types that do not map naturally to one of the other body types.

# Headers

Headers associate a fixed set of header field names with values. Clients and providers use headers to identify and route messages.

Programs can access headers as .NET properties of the message object.

Table 3 Message Header Names (Sheet 1 of 5)

Header	Description
<b>JMS Headers</b>	
These .NET properties correspond to message headers defined in the JMS specification.	
Programs can get all supported message header properties (see Message—Get Properties on page 32).	
Programs can effectively set only three message header properties—ReplyTo, CorrelationID and MsgType (see Message—Set Properties on page 34). For all other header properties, the provider ignores or overwrites values set by client programs.	
CorrelationID	<div><code>string {get; set;}</code></div> <p>Correlation ID refers to a related message. For example, when a consumer responds to a request message by sending a reply, it can set the correlation ID of the reply to indicate the request message.</p> <p>The JMS specification allows three categories of values for the correlation ID property:</p> <ul style="list-style-type: none"><li><b>Message ID</b> A message ID is a unique string that the provider assigns to a message. Programs can use these IDs to correlate messages. For example, a program can link a response to a request by setting the correlation ID of a response message to the message ID of the corresponding request message. (See also MessageID on page 14.)</li></ul> <p>Message ID strings begin with the prefix ID: (which is reserved for this purpose).</p> <ul style="list-style-type: none"><li><b>String</b> Programs can also correlate messages using arbitrary strings, with semantics determined by the application.</li></ul> <p>These strings must <i>not</i> begin with the prefix ID: (which is reserved for message IDs).</p> <ul style="list-style-type: none"><li><b>Byte Array</b> This implementation does not support byte array values for the correlation ID property. The JMS specification does not require support.</li></ul>

Table 3 Message Header Names (Sheet 2 of 5)

Header	Description
CorrelationIDAsBytes	<p>byte[] {get; set;}</p> <p>The JMS specification describes this optional utility, but EMS does <i>not</i> support it. Attempting to access this header results in <code>System.InvalidOperationException</code>.</p>
DeliveryMode	<p>int {get; set;}</p> <p>This header instructs the server concerning persistent storage for the message.</p> <p>Sending calls record the delivery mode for each message, based on either a property of the producer (<code>DeliveryMode</code> on page 90), or on a parameter to the sending call.</p> <p>For values, see the class <code>DeliveryMode</code> on page 190.</p>
MsgDeliveryMode	<p>MessageDeliveryMode {get; set;}</p> <p>This parallel .NET property accesses the same header using enumerated values (instead of ordinary integers). We recommend it over the ordinary integer-valued accessor, because it enables .NET to do stronger type checking at compile time, which can enhance program reliability.</p> <p>For values, see the class <code>MessageDeliveryMode</code> on page 194.</p>
Destination	<p>Destination {get; set;}</p> <p>Sending calls record the destination (queue or topic) of the message in this header (ignoring and overwriting any existing value). The value is based on either a property of the producer (<code>Destination</code> on page 91), or on a parameter to the send call.</p> <p>Listeners that consume messages from several destinations can use this property to determine the actual destination of a message.</p>

Table 3 Message Header Names (Sheet 3 of 5)

Header	Description
Expiration	<div><div>long {get; set;}</div><div><p>Sending calls record the expiration time (in milliseconds) of the message in this field:</p><ul style="list-style-type: none"><li>• If the time-to-live is non-zero, the expiration is the sum of that time-to-live and the sending client’s current time (GMT).</li><li>• If the time-to-live is zero, then expiration is also zero—indicating that the message never expires.</li></ul><p>The server discards a message when its expiration time has passed. However, the JMS specification does not guarantee that clients do not receive expired messages.</p><p>See <code>TimeToLive</code> on page 92.</p></div></div>
MessageID	<div><div>string {get; set;}</div><div><p>Sending calls assign a unique ID to each message, and record it in this header.</p><p>All message ID values start with the 3-character prefix <code>ID:</code> (which is reserved for this purpose).</p><p>Applications that do not require message IDs can reduce overhead costs by disabling IDs; see <code>DisableMessageID</code> on page 91. When the producer disables IDs, the value of this header is null.</p></div></div>
MsgType	<div><div>string {get; set;}</div><div><p>Some JMS providers use a message repository to store message type definitions. Client programs can store a value in this field to reference a definition in the repository. EMS supports this header, but does not use it.</p><p>The JMS specification does not define a standard message definition repository, nor does it define a naming policy for message type definitions.</p><p>Some providers require message type definitions for each application message. To ensure compatibility with such providers, client programs can set this header, even if the client application does not use it.</p><p>To ensure portability, clients can set this header with symbolic values (rather than literals), and configure them to match the provider’s repository.</p></div></div>

Table 3 Message Header Names (Sheet 4 of 5)

Header	Description
Priority	<p><code>int {get; set;}</code></p> <p>Sending calls record the priority of a message in this header, based on either a property of the producer (Priority on page 91), or on a parameter to the send call.</p> <p>The JMS specification defines ten levels of priority value, from zero (lowest priority) to 9 (highest priority). The specification suggests that clients consider 0–4 as gradations of normal priority, and priorities 5–9 as gradations of expedited priority.</p> <p>Priority affects the order in which the server delivers messages to consumers (higher values first). The JMS specification does not require all providers to implement priority ordering of messages. (EMS supports priorities, but other JMS providers might not.)</p>
Redelivered	<p><code>bool {get; set;}</code></p> <p>The server sets this header to indicate whether a message might duplicate a previously delivered message:</p> <ul style="list-style-type: none"> <li>• <code>false</code>—The server has <i>not</i> previously attempted to deliver this message to the consumer.</li> <li>• <code>true</code>—It is likely (but not guaranteed) that the server has previously attempted to deliver this message to the consumer, but the consumer did not return timely acknowledgement.</li> </ul> <p>See also, <code>SessionMode</code> on page 168.</p>
ReplyTo	<p><code>Destination {get; set;}</code></p> <p>Sending clients can set this header to request that recipients reply to the message:</p> <ul style="list-style-type: none"> <li>• When the value is a destination object, recipients can send replies to that destination. Such a message is called a <i>request</i>.</li> <li>• When the value is null, the sender does not expect a reply.</li> </ul> <p>When sending a reply, clients can refer to the corresponding request by setting the <code>CorrelationID</code> field.</p>

Table 3 Message Header Names (Sheet 5 of 5)

Header	Description
Timestamp	<div><div>long {get; set;}</div><div><p>Sending calls record a UTC timestamp in this header, indicating the approximate time that the server accepted the message.</p><p>The value is in milliseconds since January 1, 1970 (as in Java).</p><p>Applications that do not require timestamps can reduce overhead costs by disabling timestamps; see <code>DisableMessageTimestamp</code> on page 91.</p><p>When the producer disables timestamps, the value of this header is zero.</p></div></div>

# Properties

Properties associate an extensible set of property field names with values. The EMS server uses properties to attach ancillary information to messages.

Client applications can also use properties—for example, to customize message filtering; see Message Selectors on page 20.

## Setting Message Properties

Property names must conform to the syntax for message selector identifiers; see Identifiers on page 20.

Property values must *not* be null, nor the empty string.

Sending programs can set property values before sending a message.

Receiving programs cannot ordinarily set property values on inbound messages. However, the `clearProperties` method removes all existing the properties from a message, and lets the program set property values.

## EMS Properties

The JMS specification reserves the property name prefix `JMS_vendor_name_` for provider-specific properties (for EMS, this prefix is `JMS_TIBCO_`). Properties that begin with this prefix refer to features of EMS; client programs may use these properties to access those features, but not for communicating application-specific information among client programs.

Table 4 Message Property Names (Sheet 1 of 2)

Property	Description
JMS_TIBCO_CM_PUBLISHER	Correspondent name of an RVCМ sender for messages imported from TIBCO Rendezvous.
JMS_TIBCO_CM_SEQUENCE	Sequence number of an RVCМ message imported from TIBCO Rendezvous.
JMS_TIBCO_COMPRESS	Senders may set this property to request that EMS compress the message before sending it to the server. The .NET client API does not support this feature at this time.

Table 4 Message Property Names (Sheet 2 of 2)

Property	Description
JMS_TIBCO_DISABLE_SENDER	Senders may set this property to prevent the EMS server from including the sender name in the message when the server sends it to consumers; see JMS_TIBCO_SENDER.
JMS_TIBCO_IMPORTED	When the EMS server imports a message from an external message service (such as TIBCO Rendezvous or TIBCO SmartSockets), it sets this property to <code>true</code> .
JMS_TIBCO_MSG_EXT	<p>Producers can set this property to <code>true</code> to indicate that a message uses EMS extensions to the JMS specification for messages.</p> <p>The server sets this property to <code>true</code> when importing a message from an external message service, since the message might use those extensions.</p>
JMS_TIBCO_MSG_TRACE	<p>When a producer sets this property, the EMS server generates trace output when the message arrives from the producer, and whenever a consumer receives it.</p> <ul style="list-style-type: none"> <li>• When the property value is <code>null</code>, the trace output contains the message ID and sequence number.</li> <li>• When the property value is <code>body</code>, the trace output includes the message body as well.</li> </ul>
JMS_TIBCO_PRESERVE_UNDELIVERED	When this property is <code>true</code> , the server preserves a record of undeliverable messages by delivering them to the undelivered message queue, <code>\$sys.undelivered</code> .
JMS_TIBCO_SENDER	The EMS server fills this property with the <i>user name</i> (string) of the client that sent the message. This feature applies only when the <code>sender_name</code> property of the message's destination is non-null. The sender can disable this feature (overriding the destination property <code>sender_name</code> ) by setting a non-null value for the message property JMS_TIBCO_DISABLE_SENDER.
JMS_TIBCO_SS_SENDER	When the EMS server imports a message from TIBCO SmartSockets, it sets this property to the SmartSockets sender header field (in SmartSockets syntax).



## JMS Properties

The JMS specification reserves the property name prefix `JMSX` for properties defined by JMS. Client programs may use these properties to access those features, but not for communicating application-specific information among client programs.

To determine the set of JMS properties that a connection supports, call the method `JMSXPropertyNames` on page 121. For information about these properties, see the JMS specification.

## Message Selectors

---

A message selector is string that lets a client program specify a set of messages, based on the values of message headers and properties. A selector *matches* a message if, after substituting header and property values from the message into the selector string, the string evaluates to `true`. Consumers can request that the server deliver only those messages that match a selector.

The syntax of selectors is based on a subset of SQL92 conditional expression syntax.

### Identifiers

Identifiers can refer to the values of message headers and properties, but not to the message body. Identifiers are case-sensitive.

- Basic Syntax    An identifier is a sequence of letters and digits, of any length, that begins with a letter. As in Java, the set of letters includes `_` (underscore) and `$` (dollar).
- Illegal        Certain names are exceptions, which cannot be used as identifiers. In particular, `NULL`, `TRUE`, `FALSE`, `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, `IS`, and `ESCAPE` are defined to have special meaning in message selector syntax.
- Value         Identifiers refer either to message header names or property names. The type of an identifier in a message selector corresponds to the type of the header or property value. If an identifier refers to a header or property that does not exist in a message, its value is `NULL`.

### Literals

- String Literal    A string literal is enclosed in single quotes. To represent a single quote within a literal, use two single quotes; for example, `'literal' 's'`. String literals use the Unicode character encoding. String literals are case sensitive.
- Exact Numeric Literal    An exact numeric literal is a numeric value without a decimal point, such as `57`, `-957`, and `+62`; numbers in the range of `long` are supported.
- Approximate Numeric Literal    An approximate numeric literal is a numeric value with a decimal point (such as `7.`, `-95.7`, and `+6.2`), or a numeric value in scientific notation (such as `7E3` and `-57.9E2`); numbers in the range of `double` are supported. Approximate literals use the floating-point literal syntax of the Java programming language.
- Boolean Literal    The boolean literals are `TRUE` and `FALSE` (case insensitive).

Internal computations of expression values use a 3-value boolean logic similar to SQL. However, the final value of an expression is always either `TRUE` or `FALSE`—never `UNKNOWN`.

## Expressions

Selectors as Expressions	Every selector is a conditional expression. A selector that evaluates to <code>true</code> matches the message; a selector that evaluates to <code>false</code> or <code>unknown</code> does not match.
Arithmetic Expression	Arithmetic expressions are composed of numeric literals, identifiers (that evaluate to numeric literals), arithmetic operations, and smaller arithmetic expressions.
Conditional Expression	Conditional expressions are composed of comparison operations, logical operations, and smaller conditional expressions.
Order of Evaluation	Order of evaluation is left-to-right, within precedence levels. Parentheses override this order.

## Operators

Case Insensitivity	Operator names are case-insensitive.
Logical Operators	Logical operators in precedence order: <code>NOT</code> , <code>AND</code> , <code>OR</code> .
Comparison Operators	<p>Comparison operators: <code>=</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>&lt;</code>, <code>&lt;=</code>, <code>&lt;&gt;</code> (not equal).</p> <p>These operators can compare only values of comparable types. (Exact numeric values and approximate numerical values are comparable types.) Attempting to compare incomparable types yields <code>false</code>. If either value in a comparison evaluates to <code>NULL</code>, then the result is <code>unknown</code> (in SQL 3-valued logic).</p> <p>Comparison of string values is restricted to <code>=</code> and <code>&lt;&gt;</code>. Two strings are equal if and only if they contain the same sequence of characters.</p> <p>Comparison of boolean values is restricted to <code>=</code> and <code>&lt;&gt;</code>.</p>
Arithmetic Operators	<p>Arithmetic operators in precedence order:</p> <ul style="list-style-type: none"> <li>• <code>+</code>, <code>-</code> (unary)</li> <li>• <code>*</code>, <code>/</code> (multiplication and division)</li> <li>• <code>+</code>, <code>-</code> (addition and subtraction)</li> </ul> <p>Arithmetic operations obey numeric promotion rules of the Java programming language.</p>

**Between Operator**     *arithmetic-expr1* [NOT] BETWEEN *arithmetic-expr2* AND *arithmetic-expr3*

The BETWEEN comparison operator includes its endpoints. For example:

- `age BETWEEN 5 AND 9` is equivalent to `age >= 5 AND age <= 9`
- `age NOT BETWEEN 5 AND 9` is equivalent to `age < 5 OR age > 9`

**String Set Membership**     *identifier* [NOT] IN (*string-literal1*, *string-literal2*, ...)

The *identifier* must evaluate to either a string or NULL. If it is NULL, then the value of this expression is unknown.

**Pattern Matching**     *identifier* [NOT] LIKE *pattern-value* [ESCAPE *escape-character*]

The *identifier* must evaluate to a string.

The *pattern-value* is a string literal, in which some characters bear special meaning:

- `_` (underscore) can match any single character.
- `%` (percent) can match any sequence of zero or more characters.
- *escape-character* preceding either of the special characters changes them into ordinary characters (which match only themselves).

**Null Header or Property**     *identifier* IS NULL

This comparison operator tests whether a message header is null, or a message property is absent.

*identifier* IS NOT NULL

This comparison operator tests whether a message header or message property is non-null.

## White Space

White space is any of the characters space, horizontal tab, form feed, or line terminator—or any contiguous run of characters in this set.

## Data Type Conversion

Table 5 summarizes legal datatype conversions. The symbol X in Table 5 indicates that a value written into a message as the row type can be extracted as the column type. This table applies to all message values—including map pairs, headers and properties—except as noted below.

Table 5 Data Type Conversion

	bool	byte	short	char	int	long	float	double	string	byte[]
bool	X								X	
byte		X	X		X	X			X	
short			X		X	X			X	
char				X					X	
int					X	X			X	
long						X			X	
float							X	X	X	
double								X	X	
string	X	X	X		X	X	X	X	X	
byte[]										X

- Notes
- Message properties cannot have byte array values.
  - Values written as strings can be extracted as a numeric or boolean type only when it is possible to parse the string as a number of that type.

# Message

Class

- Declaration**`public class Message : ICloneable`
- Subclasses**`BytesMessage, MapMessage, ObjectMessage, StreamMessage, TextMessage`
- Purpose**Messages carry information among EMS client programs.
- Remarks**All other message types extend this root class.

Method	Description	Page
<code>Message.Acknowledge</code>	Acknowledge messages.	27
<code>Message.ClearBody</code>	Clear the body of a message.	28
<code>Message.ClearProperties</code>	Clear the properties of a message.	29
<code>Message.Clone</code>	Create a copy of the message object.	30
<code>Message.GetDeliveryModeName</code>	Convert a delivery mode constant to a string.	31
<code>Message—Get Properties</code>	Get the value of a message property.	32
<code>Message.PropertyExists</code>	Test whether a named property has been set on a message.	33
<code>Message—Set Properties</code>	Set the value of a message property.	34
<code>Message.ToString</code>	Return a string representation of a message object.	35

Field	Description
Constants defined as .NET fields.	
DEFAULT_DELIVERY_MODE	<code>int</code> <code>DeliveryMode.PERSISTENT</code>  When neither the sending call nor the producer supplies a delivery mode, this default applies.
DEFAULT_MSG_DELIVERY_MODE	<code>MessageDeliveryMode</code> <code>MessageDeliveryMode.Persistent</code>  Enumerated version of the same value.
DEFAULT_PRIORITY	<code>int</code> <code>4</code>  When neither the sending call nor the producer supplies a priority, this default applies.  See also <code>Priority</code> on page 15.
DEFAULT_TIME_TO_LIVE	<code>long</code> <code>0</code>  When neither the sending call nor the producer supplies a priority, this default applies. The default value, zero, indicates that messages do not expire.  See also <code>Expiration</code> on page 14.

**JMS Headers as .NET Properties** This table lists the JMS headers that are available as .NET properties of message objects. For details, see `Headers` on page 12.

(Sheet 1 of 2)

Header
<code>CorrelationID</code>
<code>CorrelationIDAsBytes</code>
<code>DeliveryMode</code>
<b><code>MsgDeliveryMode</code></b>
<code>Destination</code>
<code>Expiration</code>

(Sheet 2 of 2)

Header
MessageID
MsgType
Priority
Redelivered
ReplyTo
Timestamp



# Message.Acknowledge

## Method

<b>Declaration</b>	<code>virtual void Acknowledge();</code>
<b>Purpose</b>	Acknowledge messages.
<b>Remarks</b>	<p>The behavior of this call depends on the acknowledgement mode of the <i>Session</i>.</p> <ul style="list-style-type: none"> <li>• In <i>ClientAcknowledge</i> mode, this call acknowledges <i>all</i> messages that the program has consumed within the <i>session</i>. (This behavior complies with the JMS specification.)</li> <li>• In <i>ExplicitClientAcknowledge</i> mode, this call acknowledges <i>only</i> the individual message. (This mode and behavior are proprietary extensions, specific to TIBCO EMS.)</li> <li>• In <i>ExplicitClientDupsOkAcknowledge</i> mode, this call lazily acknowledges <i>only</i> the individual message. <i>Lazy</i> means that the provider client library can delay transferring the acknowledgement to the server until a convenient time; meanwhile the server might redeliver the message. (This mode and behavior are proprietary extensions, specific to TIBCO EMS.)</li> <li>• In all other modes, this call has no effect. In particular, modes that specify transactions or implicit acknowledgement do not require the consuming program to call this method. However, calling it does not produce an exception. (This behavior complies with the JMS specification.)</li> </ul>
<b>Consumed</b>	<p>Three events mark a message as <i>consumed</i>—that is, eligible for acknowledgment using this method:</p> <ul style="list-style-type: none"> <li>• Just before the provider raises an <i>EMSMessageHandler</i> event, it marks the message argument as consumed.</li> <li>• Just before the provider calls an <i>IMessageListener.OnMessage</i> callback, it marks the message argument as consumed.</li> <li>• Just before a receive call returns a message, it marks that message as consumed.</li> </ul>
<b>Redelivery</b>	The server might redeliver unacknowledged messages.
<b>See Also</b>	<p><i>MessageConsumer.Receive</i> on page 79</p> <p><i>Session</i> on page 142</p> <p><i>AcknowledgeMode</i> on page 143</p> <p><i>SessionAcknowledgeMode</i> on page 143</p> <p><i>SessionMode</i> on page 168</p>

## Message.ClearBody

---

### *Method*

<b>Declaration</b>	<code>virtual void ClearBody();</code>
<b>Purpose</b>	Clear the body of a message.
<b>Remarks</b>	<p>Clearing the body of a message leaves its header and property values unchanged.</p> <p>If the message body was read-only, this method makes it writeable. The message body appears and behaves identically to an empty body in a newly created message.</p>

## Message.ClearProperties

---

### *Method*

**Declaration**    `virtual void ClearProperties();`

**Purpose**        Clear the properties of a message.

**Remarks**     Clearing the property values of a message leaves its header values and body unchanged.

## Message.Clone

---

### *Method*

<b>Declaration</b>	<code>virtual Object Clone();</code>
<b>Implements</b>	<code>ICloneable.Clone</code>
<b>Purpose</b>	Create a copy of the message object.

# Message.GetDeliveryModeName

Method

**Declaration**     `static string GetDeliveryModeName(  
                    MessageDeliveryMode deliveryMode );`  
  
                    `static string GetDeliveryModeName  
                    (int deliveryMode );`

**Purpose**            Convert a delivery mode constant to a string.

<b>Parameters</b>	Parameter	Description
	deliveryMode	Convert this delivery mode to a string.

**Remarks**        Programs can obtain the delivery mode of a message by accessing the  
DeliveryMode header property of the message object.  
  
This method is an extension to the JMS specification.

**See Also**         DeliveryMode on page 190  
MessageDeliveryMode on page 194

# Message—Get Properties

Method

**Declaration**

```
bool GetBooleanProperty(string name);
byte GetByteProperty(string name);
short GetShortProperty(string name);
int GetIntProperty(string name);
long GetLongProperty(string name);
float GetFloatProperty(string name);
double GetDoubleProperty(string name);
string GetStringProperty(string name);
Object GetObjectProperty(string name);
```

**Purpose** Get the value of a message property.

Parameters	Parameter	Description
	name	Get the property with this name.  Property names must obey the JMS rules for a message selector identifier (see Message Selectors on page 20). Property names must not be null, and must not be empty strings.

**Remarks** The JMS specification defines eight methods to get properties with different value types—converting between compatible types. It also defines a ninth method to get values of the eight primitive types as if they were objects. All nine of these methods convert property values to the corresponding type (if possible).

When the message does not have a property set for the name, these rules apply:

- GetStringProperty and GetObjectProperty return null.
- GetBooleanProperty returns false.
- Numeric methods throw EMSException.

**See Also** Message—Set Properties on page 34

# Message.PropertyExists

Method

**Declaration**      `virtual bool PropertyExists(  
                         string name );`

**Purpose**            Test whether a named property has been set on a message.

**Parameters**

Parameter	Description
name	Test whether the message has a property with this name.

**Remarks**        This call returns `true` if the property has a value on the message; otherwise it returns `false`.

# Message—Set Properties

Method

**Declaration**

```
void SetBooleanProperty(string name, bool val);
void SetByteProperty(string name, byte val);
void SetShortProperty(string name, short val);
void SetIntProperty(string name, int val);
void SetLongProperty(string name, long val);
void SetFloatProperty(string name, float val);
void SetDoubleProperty(string name, double val);
void SetStringProperty(string name, string val);
void SetObjectProperty(string name, Object val);
```

**Purpose**

Set the value of a message property.

**Parameters**

Parameter	Description
name	Set a property with this name.  Property names must obey the JMS rules for a message selector identifier (see Message Selectors on page 20). Property names must not be null, and must not be empty strings.
val	Set the property to this value.

**Remarks**

The JMS specification defines eight methods to set properties with different primitive value types. It also defines a ninth method to set a property to an object representation of any of the eight primitive types.

**See Also**

Message—Get Properties on page 32



## Message.ToString

---

*Method*

<b>Declaration</b>	<code>override string ToString();</code>
<b>Purpose</b>	Return a string representation of a message object.
<b>Remarks</b>	The string includes the body type, headers (name-value pairs), properties (name-value pairs), and body content.

# BytesMessage

Class

- Declaration

class BytesMessage : Message
- Purpose

A message containing a stream of uninterpreted bytes.
- Remarks

Messages with this body type contain a single value, which is a byte stream.

Method	Description	Page
BytesMessage—Read	Read primitive datatypes from the byte stream in the message body.	37
BytesMessage.ReadBytes	Read bytes to a byte array from the byte stream in the message body.	39
BytesMessage—Write	Write primitive datatypes to the byte stream in the message body.	40
BytesMessage.WriteBytes	Write bytes from a byte array to the byte stream in the message body.	42
BytesMessage.Reset	Set the read position to the beginning of the byte stream, and mark the message body as read-only.	43

Member	Description
Properties	
BodyLength	<div>long {get;}</div> <div>Programs can get the length of the message body (in bytes). Programs cannot set this .NET property.</div>

Superclasses

Message on page 24

# BytesMessage—Read

## Method

**Declaration**

```
bool ReadBoolean();
sbyte ReadByte();
byte ReadUnsignedByte();
short ReadShort();
ushort ReadUnsignedShort();
char ReadChar();
int ReadInt();
long ReadLong();
float ReadFloat();
double ReadDouble();
String ReadUTF();
```

**Purpose** Read primitive datatypes from the byte stream in the message body.

**Remarks** The JMS specification defines these eleven methods to extract data from the byte stream of a BytesMessage.

Each call reads a unit of data from the stream, and advances the read position so that the next read call gets the next datum.

Parameter	Description
value	The method reads a datum from the message, and stores it in this location.
length	ReadUTF reads a UTF-8 string. Since the length of the string cannot be determined in advance, the method stores the actual length of the string in this location.

Table 6 BytesMessage Read Methods (Sheet 1 of 2)

Method	# Bytes	Interpret As
ReadBoolean	1	bool
ReadByte	1	sbyte
ReadUnsignedByte	1	byte
ReadShort	2	short
ReadUnsignedShort	2	ushort
ReadChar	2	char

Table 6 *BytesMessage Read Methods (Sheet 2 of 2)*

Method	# Bytes	Interpret As
ReadInt	4	int
ReadLong	8	long
ReadFloat	4	float
ReadDouble	8	double
ReadUTF	varies	String Encoded as UTF-8

**See Also** `BytesMessage.ReadBytes` on page 39

# BytesMessage.ReadBytes

Method

**Declaration**     `int ReadBytes(byte[] value);`  
`int ReadBytes(byte[] value, int length);`

**Purpose**     Read bytes to a byte array from the byte stream in the message body.

Parameters	Parameter	Description
	value	The program supplies a byte array. The call fills it with bytes from the byte stream.
	length	Read (at most) this number of bytes from the stream.  When present, the length argument must be between zero and value.length (inclusive); otherwise the call throws a <code>System.IndexOutOfRangeException</code> (and does not read any bytes).

**Remarks**     Each call reads bytes from the stream into the byte array, and advances the read position.  
  
When the program supplies a length argument, the call attempts to read length bytes; otherwise it attempts to read value.length bytes.

**Returns**     This call returns the actual number of bytes read. When the call cannot read even one byte, it returns -1.

# BytesMessage—Write

## Method

**Declaration**

```
void WriteBoolean(bool value);
void WriteByte(byte value);
void WriteShort(short value);
void WriteChar(char value);
void WriteInt(int value);
void WriteLong(long value);
void WriteFloat(float value);
void WriteDouble(double value);
void WriteUTF(string value);
void WriteObject(Object value);
```

**Purpose** Write primitive datatypes to the byte stream in the message body.

Parameter	Description
value	Write this value to the message.

**Remarks** The JMS specification defines these ten methods to insert data into the byte stream of a BytesMessage.

Each call writes a data value to the stream, and advances the write position so that the next write call appends to the new end of the stream.

Table 7 BytesMessage Write Methods (Sheet 1 of 2)

Method	# Bytes	Notes
WriteBoolean	1	
WriteByte	1	
WriteShort	2	
WriteChar	2	
WriteInt	4	
WriteLong	8	
WriteFloat	4	
WriteDouble	8	
WriteUTF	varies	Encoded as UTF-8

Table 7 BytesMessage Write Methods (Sheet 2 of 2)

Method	# Bytes	Notes
WriteObject	varies	Converts an object to a primitive value (if possible), and writes that value to the byte stream.

**See Also** BytesMessage.WriteBytes on page 42

# BytesMessage.WriteBytes

Method

**Declaration**

```
void WriteBytes(byte[] value);  
  
void WriteBytes(byte[] value, int offset, int length);
```

**Purpose**

Write bytes from a byte array to the byte stream in the message body.

**Parameters**

Parameter	Description
value	Write bytes from this byte array to the message.
offset	Begin with the byte at this offset within the byte array.
length	Write this number of bytes from the byte array.

**Remarks**

Each call writes bytes from the byte array into the stream, and advances the write position.

When the program supplies `offset` and `length` arguments, the call attempts to write the specified bytes to the stream; otherwise it attempts to write the entire byte array.

**Offset & Length**

When present, the `offset` and `length` arguments must conform to these restrictions:

- `offset` must be in the range `[0, value.length-1]`
- `length` must be in the range `[0, value.length]`
- `offset+length` must be in the range `[0, value.length]`

That is, these two arguments must specify a span of bytes within the `value` argument. Otherwise the call throws a `System.IndexOutOfRangeException` (and does not write any bytes).



# BytesMessage.Reset

---

*Method*

<b>Declaration</b>	<code>void Reset();</code>
<b>Purpose</b>	Set the read position to the beginning of the byte stream, and mark the message body as read-only.
<b>Remarks</b>	Reset prepares a message body for reading, as if the message were newly received. Contrast <code>Message.ClearBody</code> on page 28, which clears a message body in preparation for writing, as if it were newly created.

# MapMessage

Class

- Declaration

class MapMessage : Message
- Purpose

A message containing a set of name-value pairs.
- Remarks

Messages with this body type contain several values, indexed by name.

Method	Description	Page
MapMessage—Get	Get primitive datatypes from a map message.	46
MapMessage.ItemExists	Test that a named pair is set.	47
MapMessage—Set	Set a name-value pair in a map message.	48
MapMessage.SetBytes	Set a name-value pair to a byte array value.	49

Member	Description
Properties	
FieldCount	<div>int {get;}</div> <div>Programs can get the number of data items in the message body. Programs cannot set this .NET property.</div>
MapNames	<div>System.Collections.IEnumerator {get;}</div> <div>Programs can get an enumerator that produces the names of all the data items in the message body.</div>

- Superclasses

Message on page 24
- Extensions

TIBCO Enterprise Message Service extends the JMS MapMessage and StreamMessage body types in two ways. These extensions allow TIBCO Enterprise Message Service to exchange messages with TIBCO Rendezvous and TIBCO SmartSockets programs, which have certain features not available within the JMS specification.

- You can insert another `MapMessage` or `StreamMessage` instance as a submessage into a `MapMessage` or `StreamMessage`, generating a series of nested messages, instead of a flat message.
- You can use arrays as well as primitive types for the values.

These extensions add considerable flexibility to the two body types. However, they are extensions and therefore not compliant with JMS specifications. Extended messages are tagged as extensions with the vendor property tag `JMS_TIBCO_MSG_EXT`.

For more information on message compatibility with Rendezvous messages, see Message Body on page 87 in *TIBCO Enterprise Message Service Release Notes*.

# MapMessage—Get

Method

**Declaration**

```
bool GetBoolean(string name);
byte GetByte(string name);
short GetShort(string name);
char GetChar(string name);
int GetInt(string name);
long GetLong(string name);
float GetFloat(string name);
double GetDouble(string name);
string GetString(string name);
byte[] GetBytes(string name);
Object GetObject(string name);
```

**Purpose** Get primitive datatypes from a map message.

<b>Parameters</b>	Parameter	Description
	name	Get the value associated with this name.

**Remarks** The JMS specification defines eleven methods to extract data from the name-value pairs of a MapMessage. Ten of these methods extract primitive data values. The GetObject method gets these values as if they were objects.

**Returns** Each call finds the named pair (if it exists) and returns its value.  
When the message does not have a field set for the name, these calls return null.

# MapMessage.ItemExists

Method

**Declaration**     `bool ItemExists(string name);`

**Purpose**     Test that a named pair is set.

**Parameters**

Parameter	Description
name	Test for a pair with this name.

# MapMessage—Set

Method

Declaration

```
void SetBoolean(string name, bool value);
void SetByte(string name, byte value);
void SetShort(string name, short value);
void SetChar(string name, char value);
void SetInt(string name, int value);
void SetLong(string name, long value);
void SetFloat(string name, float value);
void SetDouble(string name, double value);
void SetString(string name, String value);
void SetObject(string name, Object value);
```

Purpose

Set a name-value pair in a map message.

Parameters

Parameter	Description
name	Set the pair with this name.  Field names must not be null, and must not be empty strings.
value	Associate this value with the name.

Remarks

The JMS specification defines eleven methods to set name-value pairs in a `MapMessage` (these ten, plus `MapMessage.SetBytes` on page 49).

The first nine of these methods set pairs with primitive value types. The `SetObject` method first converts an object to a primitive type (if possible), and then places that value in the pair.

See Also

`MapMessage.SetBytes` on page 49

# MapMessage.SetBytes

## Method

**Declaration**     `void SetBytes(string name, byte[] value);`  
                          `void SetBytes(string name, byte[] value, int offset, int length);`

**Purpose**     Set a name-value pair to a byte array value.

## Parameters

Parameter	Description
name	Set the pair with this name.
value	Associate bytes from this byte array as the value of the name.
offset	Begin with the byte at this offset within the byte array.
length	Write this number of bytes from the byte array.

**Remarks**     When the program supplies `offset` and `length` arguments, the call extracts the specified bytes and uses them as the value; otherwise it uses the entire byte array.

When present, the `offset` and `length` arguments must be between zero and `value.length` (inclusive), and their sum must also fall within the same range. That is, these two arguments must specify a span of bytes within the `value` argument. Otherwise the call throws an `System.IndexOutOfRangeException` (and does not set any value).

**See Also**     [MapMessage—Set](#) on page 48

# ObjectMessage

Class

- Declaration

class ObjectMessage : Message
- Purpose

A message containing a serializable object.
- Remarks

Setting the content of a MessageObject stores a snapshot of the object. subsequent changes to the original object do not affect the message.

Member	Description
Properties	
TheObject	Object {get; set;}  Programs can get and set the object in an ObjectMessage.

Method	Description	Page
ObjectMessage	Constructor.	51

- .NET Compact Framework

Object serialization differs among the various EMS language APIs in ways that are incompatible. An ObjectMessage contains a serialized object. Therefore EMS programs can only send an ObjectMessage to another program written in the same language; for example, Java to Java, C to C, .NET to .NET, and .NET Compact Framework to .NET Compact Framework. In particular, notice that a .NET Compact Framework client and a full .NET client cannot exchange an ObjectMessage.

Furthermore, the .NET Compact Framework supports only a limited set of objects for TheObject in an ObjectMessage—namely, bool, int, long, short, double, float, byte, bytes, char, string, short[], int[], long[], float[], double[], MapMessage, StreamMessage, and program-defined classes that implement IEMSSerialziable. Attempting to set the value to an unsupported object type results in MessageFormatException. This restriction applies only to .NET Compact Framework (the full .NET EMS API is exempt).
- Superclasses

Message on page 24  
IEMSSerialziable on page 191  
MessageFormatException on page 242



# ObjectMessage

## Constructor

Declaration

```
public ObjectMessage(  
    Session session,  
    object obj );  
  
public ObjectMessage(  
    Session session );
```

Purpose

Create an object message.

Parameters

Parameter	Description
session	Associate the new message with this session.
obj	Use this object as the value of the new message. When absent, construct an empty object message.

See Also

[.NET Compact Framework on page 50](#)  
[Session.CreateObjectMessage on page 156](#)

# StreamMessage

Class

- Declaration

class StreamMessage : Message
- Purpose

A message containing a stream of data items.
- Remarks

Each datum in the stream must be a primitive type, or an object representation of a primitive type.

Member	Description	Page
StreamMessage—Read	Read primitive datatypes from a stream message.	54
BytesMessage.ReadBytes	Read a byte array from a stream message.	55
StreamMessage.Reset	Set the read position to the beginning of the stream, and mark the message body as read-only.	56
StreamMessage—Write	Write primitive datatypes to a stream message.	57
BytesMessage.WriteBytes	Write bytes from a byte array to a stream message.	58

Member	Description
Properties	
FieldCount	<div>int {get;}</div> <div>Programs can get the number of data items in the message body. Programs cannot set this property.</div>

- Superclasses

Message on page 24
- Extensions

TIBCO Enterprise Message Service extends the MapMessage and StreamMessage body types in two ways. These extensions allow TIBCO Enterprise Message Service to exchange messages with TIBCO Rendezvous and ActiveEnterprise formats that have certain features not available within the JMS specification.

- You can insert another `MapMessage` or `StreamMessage` instance as a submessage into a `MapMessage` or `StreamMessage`, generating a series of nested messages, instead of a flat message.
- You can use arrays as well as primitive types for the values.

These extensions add considerable flexibility to the two body types. However, they are extensions and therefore not compliant with JMS specifications. Extended messages are tagged as extensions with the vendor property tag `JMS_TIBCO_MSG_EXT`.

For more information on message compatibility with Rendezvous messages, see Message Body on page 87 in *TIBCO Enterprise Message Service User's Guide*.

## StreamMessage—Read

---

### Method

**Declaration**

```
bool ReadBoolean();  
sbyte ReadByte();  
short ReadShort();  
char ReadChar();  
int ReadInt();  
long ReadLong();  
float ReadFloat();  
double ReadDouble();  
String ReadString();  
Object ReadObject();
```

**Purpose** Read primitive datatypes from a stream message.

**Remarks** The JMS specification defines these methods to extract data from a `StreamMessage`. (See also `StreamMessage.ReadBytes` on page 55.)

Each call reads a unit of data from the stream, and advances the read position so that the next read call gets the next datum.

**See Also** `StreamMessage.ReadBytes` on page 55

## StreamMessage.ReadBytes

---

*Method*

**Declaration**     `int ReadBytes(byte[] value);`

**Purpose**     Read a byte array from a stream message.

**Parameters**

Parameter	Description
value	The program supplies a byte array. The call fills it with bytes from the stream message.

---

**Remarks**     Each call reads bytes from the stream into the byte array, and advances the read position.

                    This call returns the actual number of bytes read. When the call cannot read even one byte, it returns -1.

                    A program that calls this method must call it repeatedly until it returns -1, indicating that the program has extracted the complete set of bytes. Only then may the program call another read method.

## StreamMessage.Reset

---

### *Method*

**Declaration**    `void Reset();`

**Purpose**        Set the read position to the beginning of the stream, and mark the message body as read-only.

**Remarks**     Reset prepares a message body for reading, as if the message were newly received. Contrast `Message.ClearBody` on page 28, which clears a message body in preparation for writing, as if it were newly created.

## StreamMessage—Write

### Method

**Declaration**

```
void WriteBoolean(bool value);  
void WriteByte(byte value);  
void WriteShort(short value);  
void WriteChar(char value);  
void WriteInt(int value);  
void WriteLong(long value);  
void WriteFloat(float value);  
void WriteDouble(double value);  
void WriteString(string value);  
void WriteObject(Object value);
```

**Purpose** Write primitive datatypes to a stream message.

**Remarks** The JMS specification defines these methods to insert data into a `StreamMessage`. (See also `StreamMessage.WriteBytes` on page 58.)

Each call writes a data value to the stream, and advances the write position so that the next write call appends to the new end of the stream.

`WriteObject` converts an object to a primitive value (if possible), and writes that value to the stream message.

Parameter	Description
value	Write this datum.

**See Also** `StreamMessage.WriteBytes` on page 58

# StreamMessage.WriteBytes

Method

**Declaration**

```
void WriteBytes(byte[] value);  
  
void WriteBytes(byte[] value, int offset, int length);
```

**Purpose**

Write bytes from a byte array to a stream message.

**Parameters**

Parameter	Description
value	Write bytes from this byte array to the message.
offset	Begin with the byte at this offset within the byte array.
length	Write this number of bytes from the byte array.

**Remarks**

Each call writes bytes from the byte array into the stream, and advances the write position.

When the program supplies `offset` and `length` arguments, the call attempts to write the specified bytes to the stream; otherwise it attempts to write the entire byte array.

When present, the `offset` and `length` arguments must be between zero and `value.Length` (inclusive), and their sum must also fall within the same range. That is, these two arguments must specify a span of bytes within the `value` argument. Otherwise the call throws an `IndexOutOfRangeException` (and does not write any bytes).



# TextMessage

Class

- Declaration**`class TextMessage : Message`
- Purpose**A message containing a text string.
- Remarks**Messages with this body type contain a single value, which is a string.

Member	Description
Properties	
Text	<code>string {get; set;}</code> Programs can get and set the text string in a TextMessage.

Method	Description	Page
TextMessage	Constructor.	60

**Superclasses** Message on page 24

# TextMessage

## Constructor

Declaration

```
public TextMessage(  
    Session session,  
    string text );  
  
public ObjectMessage(  
    Session session );
```

Purpose

Create a text message.

Parameters

Parameter	Description
session	Associate the new message with this session.
text	Use this string as the value of the new message.

See Also

`Session.CreateTextMessage` on page 162

## Chapter 4      **Destination**

### Topics

---

- *Destination Overview, page 62*
- *Destination, page 65*
- *Queue, page 66*
- *TemporaryQueue, page 68*
- *TemporaryTopic, page 70*
- *Topic, page 72*

## Destination Overview

Destination objects represent destinations within the EMS server—the queues and topics to which programs send messages, and from which they receive messages. Queues deliver each message to exactly one consumer. Topics deliver each message to every subscriber. Queues and topics can be static, dynamic or temporary.

Table 8 Destination Overview (Sheet 1 of 3)

Aspect	Static	Dynamic	Temporary
Purpose	Static destinations let administrators configure EMS behavior at the enterprise level. Administrators define these administered objects, and client programs use them—relieving program developers and end users of the responsibility for correct configuration.	Dynamic destinations give client programs the flexibility to define destinations as needed for short-term use.	Temporary destinations are ideal for limited-scope uses, such as reply subjects.
Scope of Delivery	Static destinations support concurrent use. That is, several client processes (and in several threads within a process) can create local objects denoting the destination, and consume messages from it.	Dynamic destinations support concurrent use. That is, several client processes (and in several threads within a process) can create local objects denoting the destination, and consume messages from it.	Temporary destinations support only local use. That is, only the client connection that created a temporary destination can consume messages from it. However, servers connected by routes do exchange messages sent to temporary topics.

Table 8 Destination Overview (Sheet 2 of 3)

Aspect	Static	Dynamic	Temporary
Creation	Administrators create static destinations using EMS server administration tools or API.	<p>If the server configuration permits dynamic destinations, client programs can create one in two steps:</p> <ol style="list-style-type: none"> <li>1. Create a local destination object; see <i>Session</i> on page 142.</li> <li>2. Send a message to that destination, or create a consumer for it. Either of these actions automatically creates the destination in the server.</li> </ol>	Client programs create temporary destinations; see <i>Session</i> on page 142.
Lookup	Client programs lookup static destinations by name. Successful lookup returns a local object representation of the destination; see <code>LookupContext.Lookup</code> on page 186.	Client programs lookup dynamic destinations by name. Successful lookup returns a local object representation of the destination; see <code>LookupContext.Lookup</code> on page 186.	Not applicable.

Table 8 Destination Overview (Sheet 3 of 3)

Aspect	Static	Dynamic	Temporary
Duration	A static destination remains in the server until an administrator explicitly deletes it.	<div>A dynamic destination remains in the server as long as at least one client actively uses it. The server automatically deletes it (at a convenient time) when all applicable conditions are true:<ul style="list-style-type: none"><li>• <b>Topic or Queue</b> all client programs that access the destination have disconnected</li><li>• <b>Topic</b> no offline durable subscribers exist for the topic</li><li>• <b>Queue</b> queue, no messages are stored in the queue</li></ul></div>	A temporary destination remains in the server either until the client that created it explicitly deletes it, or until the client disconnects from the server.

# Destination

---

## Class

<b>Declaration</b>	<code>class Destination</code>
<b>Purpose</b>	Root behavior of all destinations.
<b>Remarks</b>	<p>Administrators define destinations in the server. Client programs access them using methods of <code>LookupContext</code>.</p> <p>Programs do not create instances of this class; instead, they create instances of its subclasses.</p>
<b>Subclasses</b>	<p>Queue on page 66</p> <p>TemporaryQueue on page 68</p> <p>Topic on page 72</p> <p>TemporaryTopic on page 70</p>
<b>See Also</b>	LookupContext on page 182

# Queue

Class

- Declaration**`class Queue : Destination`
- Purpose**Queues deliver each message to exactly one consumer.

Member	Description
Properties	
QueueName	<code>string {get;}</code> The lookup name of the queue object. Each queue has a name that is unique among all queues.

Method	Description	Page
Queue	Constructor.	67

- Subclasses**TemporaryQueue on page 68



# Queue

## Constructor

**Declaration**     `Queue(  
                  string name);`

**Purpose**            Create a queue object.

**Remarks**        This constructor creates only local objects (within the program). It does not attempt to lookup the corresponding server object until the program creates a `MessageConsumer` or a `MessageProducer` that uses the queue. That automatic lookup can result in either of two outcomes:

- If lookup succeeds, it binds the local queue object to the server queue object.
- If lookup fails, the server creates a new dynamic queue.

Parameter	Description
name	Find or create a queue with this name.

**See Also**        `Session.CreateQueue` on page 158  
                  `LookupContext` on page 182

# TemporaryQueue

Class

- Declaration

class TemporaryQueue : Queue
- Purpose

Programs can use temporary queues as reply destinations.
- Remarks

Programs create temporary queues using `Session.CreateTemporaryQueue`.

A temporary queue exists only for the duration of the session’s connection, and is available only within that connection.

Only consumers associated with the same connection as the temporary queue can consume messages from it.

Method	Description	Page
<code>TemporaryQueue.Delete</code>	Delete a temporary queue.	69

**See Also**    `Session.CreateTemporaryQueue` on page 160

## TemporaryQueue.Delete

---

### Method

**Declaration**    `void Delete();`

**Purpose**        Delete a temporary queue.

**Remarks**     When a client deletes a temporary queue, the server deletes any unconsumed messages in the queue.

                  If the client still has listeners or receivers for the queue, or is in the middle of a `Receive` call, then `Delete` throws an `EMSException`.

# TemporaryTopic

Class

- Declaration

class TemporaryTopic : Topic
- Purpose

Programs can use temporary topics as reply destinations.
- Remarks

Programs create temporary topics using `Session.CreateTemporaryTopic`.

A temporary topic exists only for the duration of the session’s connection, and is available only within that connection.

Only consumers associated with the same connection as the temporary topic can consume messages from it.

Servers connected by routes do exchange messages sent to temporary topics.

Method	Description	Page
<code>TemporaryTopic.Delete</code>	Delete a temporary topic.	71

**See Also**    `Session.CreateTemporaryTopic` on page 161

## TemporaryTopic.Delete

---

*Method*

**Declaration**    `void Delete();`

**Purpose**        Delete a temporary topic.

**Remarks**     When a client deletes a temporary topic, the server deletes any unconsumed messages in the topic.

                  If the client still has listeners or receivers for the topic, or is in the middle of a `Receive` call, then `Delete` throws an `EMSEException`.

# Topic

Class

- Declaration

class Topic : Destination
- Purpose

Topics deliver each message to multiple consumers.

Member	Description
Properties	
TopicName	<div>string {get;}</div> <div>The lookup name of the topic object. Each topic has a name that is unique among all topics.</div>

Method	Description	Page
Topic	Constructor.	73

- Subclasses

TemporaryTopic on page 70

# Topic

## Constructor

<b>Declaration</b>	<pre> Topic(     string name ); </pre>
<b>Purpose</b>	Create a topic object.
<b>Remarks</b>	<p>This constructor creates only local objects (within the program). It does not attempt to lookup the corresponding server object until the program creates a <code>MessageConsumer</code> or a <code>MessageProducer</code> that uses the topic. That automatic lookup can result in either of two outcomes:</p> <ul style="list-style-type: none"> <li>• If lookup succeeds, it binds the local topic object to the server topic object.</li> <li>• If lookup fails, the server creates a new dynamic topic.</li> </ul>

Parameter	Description
name	Find or create a topic with this name.

<b>See Also</b>	<a href="#">Session.CreateTopic</a> on page 163 <a href="#">LookupContext</a> on page 182
-----------------	--





## Chapter 5      **Consumer**

Each message consumer receives messages from a destination.

### Topics

---

- *MessageConsumer*, page 76
- *QueueReceiver*, page 81
- *TopicSubscriber*, page 82

# MessageConsumer

Class

Declaration	class MessageConsumer
Purpose	Root behavior of all consumers.
Remarks	<p>Consumers can receive messages synchronously (using the Receive methods), or asynchronously.</p> <p>Consumers can receive messages asynchronously in either of two idioms. Programmers may select either idiom—but not both (which would cause duplicate message processing, with undefined behavior).</p> <ul style="list-style-type: none"><li>• MessageHandler events let programs receive messages in a .NET programming idiom.</li><li>• In contrast, the MessageListener property mimics the way in which JMS provides similar functionality in a Java programming idiom.</li></ul>
Subclasses	QueueReceiver TopicSubscriber

(Sheet 1 of 2)

Member	Description
Events	
MessageHandler	<p>EMSMessageHandler</p> <p>The client library raises an event when a message arrives at the destination. The program implements a handler delegate to processes it asynchronously, and registers the delegate here. See Remarks, above.</p>
Properties	
MessageListener	<p>IMessageListener {get; set;}</p> <p>When a message arrives, the client library calls this listener’s onMessage method with the message as its argument. The program implements the message listener interface, and registers a message listener object by setting this property. See Remarks, above.</p>

(Sheet 2 of 2)

Member	Description
MessageSelector	<p>string {get;}</p> <p>A message selector restricts the set of messages that the consumer receives to those that match the selector; see Message Selectors on page 20.</p> <p>Programs can set this property only when creating the consumer object; see Session.CreateConsumer on page 152.</p>

Method	Description	Page
MessageConsumer.Close	Stop receiving messages; reclaim resources.	78
MessageConsumer.Receive	Receive a message (synchronous).	79
MessageConsumer.ReceiveNoWait	Receive a message (synchronous, non-blocking).	80

**See Also**    [IMessageListener](#) on page 86  
[EMSMessageHandler](#) on page 83

## MessageConsumer.Close

---

### Method

**Declaration**    `void Close();`

**Purpose**        Stop receiving messages; reclaim resources.

**Remarks**     If a receive call or a message listener is in progress, then `Close` waits until that call returns.

Message consumers rely on resources outside the client program. To reclaim these resources in a timely manner, we recommend that client programs explicitly close message consumer objects (rather than waiting for garbage collection).

**See Also**      `MessageConsumer.Receive` on page 79  
                 `IMessageListener` on page 86  
                 `Session.CreateConsumer` on page 152

# MessageConsumer.Receive

Method

**Declaration**     `Message Receive();`  
                          `Message Receive(`  
                              `long timeout );`

**Purpose**     Receive a message (synchronous).

Parameter	Description
timeout	When present, wait no longer than this interval (in milliseconds) for a message to arrive. Zero is a special value, which specifies no timeout (block indefinitely).  When absent, the default value is zero.

**Remarks**     This method consumes the next message from the destination.

When the destination does not have any messages ready, this method blocks:

- If a message arrives at the destination, this call immediately returns that message.
- If the (non-zero) timeout elapses before a message arrives, this call returns null.
- If another thread closes the consumer, this call returns null.

When calling receive within a transaction, the consumer retains the message until transaction commits.

## MessageConsumer.ReceiveNoWait

---

### *Method*

**Declaration**     `Message ReceiveNoWait();`

**Purpose**            Receive a message (synchronous, non-blocking).

**Remarks**        When the destination has at least one message ready, this method immediately returns the next message.

When the destination does *not* have any messages ready, this method immediately returns null.

When calling receive within a transaction, the consumer retains the message until transaction commits.

# QueueReceiver

Class

- Declaration**`class QueueReceiver : MessageConsumer`
- Purpose**Consume messages from a queue.
- Remarks**This class inherits almost all of its behavior from `MessageConsumer`. It adds only a property, specializing the generic destination to a queue.

Member	Description
Properties	
Queue	<div><code>Queue {get;}</code></div> <div>The receiver consumes messages from this queue.</div> <div>Programs set this queue when creating the receiver, and cannot subsequently change it.</div>

# TopicSubscriber

Class

- Declaration

class TopicSubscriber : MessageConsumer
- Purpose

Consume messages from a topic.
- Remarks

This class inherits almost all of its behavior from MessageConsumer; it adds only two properties.

Member	Description
Properties	
NoLocal	<div>bool {get;}</div> <div>When true, the subscriber does not receive messages sent through the same server connection (that is, the connection associated with the subscriber).</div> <div>Programs set this property when creating the subscriber, and cannot subsequently change it.</div>
Topic	<div>Topic {get;}</div> <div>The subscriber consumes messages from this topic.</div> <div>Programs set this topic property when creating the subscriber, and cannot subsequently change it.</div>



# EMSMessageHandler

## Delegate

Declaration	<code>delegate void EMSMessageHandler(     object sender,     EMSMessageEventArgs args );</code>
Purpose	Asynchronously process an arriving message.
Remarks	<p>This delegate provides an asynchronous pathway for receiving messages. The program implements this delegate, and registers it with a <code>MessageConsumer</code>. When a message arrives, the client library raises an event. This delegate processes the event, which presents the message.</p> <p><code>EMSMessageHandler</code> receives messages in a .NET programming idiom. In contrast, <code>IMessageListener</code> mimics the way in which JMS provides similar functionality in a Java programming idiom. Programmers may select either idiom—but not both (which would cause duplicate message processing, with undefined behavior).</p>

Parameter	Description
sender	The <code>MessageConsumer</code> object that raised a message event.
args	The event, which contains the message object.

### Example 1 Message Event Handler

```
...
consumer.MessageHandler += new EMSMessageHandler(handleMsg);
...
private void handleMsg(object sender, EMSMessageEventArgs arg)
{
    Message m = arg.Message;
    Console.WriteLine("Received message: " + m);
    ...
}
```

Serialization	In compliance with the JMS specification, sessions distribute messages to listeners and event handler delegates in serial (non-concurrent) fashion.
See Also	<a href="#">MessageConsumer</a> on page 76 <a href="#">EMSMessageEventArgs</a> on page 84

# EMSMessageEventArgs

Class

- Declaration

class EMSMessageEventArgs : EventArgs
- Purpose

Present an arriving message as a .NET event.
- Remarks

EMSMessageHandler delegates receive this object as an argument.

Member	Description
Properties	
Message	Message {get;}  Programs can get the message that triggered the event.

Method	Description	Page
EMSMessageEventArgs	Constructor.	85

# EMSMessageEventArgs

Constructor

Declaration

EMSMessageEventArgs(  
    Message msg );

Purpose

Create a message event.

Parameter	Description
msg	The new event encapsulates this message, and signals its arrival.

# IMessageListener

Interface

- Declaration

interface IMessageListener
- Purpose

Asynchronously process an arriving message.
- Remarks

This interface provides an asynchronous pathway for receiving messages. The program implements this interface, and registers a message listener with a `MessageConsumer`. When a message arrives, the client library calls the listener's `onMessage` method with the message as its argument.

`IMessageListener` mimics the way in which JMS receives messages in a Java programming idiom. In contrast, `EMSMessagesHandler` provides similar functionality in a .NET idiom. Programmers may select either idiom—but not both (which would cause duplicate message processing, with undefined behavior).

Method	Description	Page
<code>IMessageListener.OnMessage</code>	Process inbound messages (asynchronous).	87

- Serialization

In compliance with the JMS specification, sessions distribute messages to listeners and event handler delegates in serial (non-concurrent) fashion.
- Deprecated

In earlier releases, clients could register listeners with sessions as well as consumers. This practice is now deprecated—we recommend migrating existing code to one of these two practices:

  - Java Idiom** Register listeners with consumers.
  - .NET Idiom** Register `EMSMessagesHandler` delegates with consumers.
- See Also

`MessageConsumer` on page 76

`Session` on page 142

# IMessageListener.OnMessage

Method

**Declaration**      `void OnMessage(  
                    Message message );`

**Purpose**            Process inbound messages (asynchronous).

Parameter	Description
message	Process this message.



## Chapter 6      **Producer**

Message producers send messages to destinations on the server.

### Topics

---

- *MessageProducer*, page 90
- *QueueSender*, page 96
- *TopicPublisher*, page 99

# MessageProducer

Class

Declaration	class MessageProducer
Purpose	Root behavior of all producers.
Remarks	<p>Clients use message producers to send messages. A message producer object can store several parameters that affect the messages it sends.</p> <p>This class lacks a constructor. Instead, clients create message producers using methods of a Session object; subclasses (such as QueueSession and TopicSession) each define methods to create corresponding producer subclasses.</p>
Subclasses	QueueSender TopicPublisher

(Sheet 1 of 3)

Member	Description
Properties	
DeliveryMode	<p>int {get; set;}</p> <p>Delivery mode instructs the server concerning persistent storage.</p> <p>Programs can use this property to define a default delivery mode for messages that this producer sends. Individual sending calls can override this default value.</p> <p>For values, see the class <code>DeliveryMode</code> on page 190.</p>
MsgDeliveryMode	<p>MessageDeliveryMode {get; set;}</p> <p>This parallel property accesses the same default value using enumerated values (instead of ordinary integers). We recommend it over the ordinary integer-valued accessor, because it enables .NET to do stronger type checking at compile time, which can enhance program reliability.</p>



(Sheet 2 of 3)

Member	Description
Destination	<p><code>Destination {get;}</code></p> <p>Each send call directs a message to a destination (queue or topic).</p> <p>This property defines a default destination for messages that this producer sends. Individual sending calls can override this default value.</p>
DisableMessageID	<p><code>bool {get; set;}</code></p> <p>Applications that do not require message IDs can reduce overhead costs by disabling IDs (set this property to <code>true</code>).</p>
DisableMessageTimestamp	<p><code>bool {get; set;}</code></p> <p>Applications that do not require timestamps can reduce overhead costs by disabling timestamps (set this property to <code>true</code>).</p>
Priority	<p><code>int {get; set;}</code></p> <p>Priority affects the order in which the server delivers messages to consumers (higher values first).</p> <p>The JMS specification defines ten levels of priority value, from zero (lowest priority) to 9 (highest priority). The specification suggests that clients consider 0–4 as gradations of normal priority, and priorities 5–9 as gradations of expedited priority.</p> <p>Programs can use this property to define a default priority for messages that this producer sends. Individual sending calls can override this default value.</p>

(Sheet 3 of 3)

Member	Description
TimeToLive	<div><div><div>long {get; set;}</div></div><div><div>Time-to-live (in milliseconds) determines the expiration time of a message.</div><div><ul style="list-style-type: none"><li>• If the time-to-live is non-zero, the expiration is the sum of that time-to-live and the sending client’s current time (GMT). This rule applies even within sessions with transaction semantics—the timer begins with the send call, not the commit call.</li><li>• If the time-to-live is zero, then expiration is also zero—indicating that the message never expires.</li></ul></div></div></div> <div><div>Programs can use this property to define a default time-to-live for messages that this producer sends. Individual sending calls can override this default value.</div><div>Whenever your application uses non-zero values for message expiration or time-to-live, you must ensure that clocks are synchronized among all the host computers that send and receive messages. Synchronize clocks to a tolerance that is a very small fraction of the smallest or time-to-live.</div></div>

Method	Description	Page
MessageProducer.Close	Destroy the producer object; reclaim resources.	93
MessageProducer.Send	Send a message.	94

# MessageProducer.Close

---

*Method*

<b>Declaration</b>	<code>void Close();</code>
<b>Purpose</b>	Destroy the producer object; reclaim resources.
<b>Remarks</b>	Message producers rely on resources outside the client program. To reclaim these resources in a timely manner, we recommend that client programs explicitly close message producer objects (rather than waiting for garbage collection).
<b>See Also</b>	<code>Session.CreateProducer</code> on page 157

# MessageProducer.Send

Method

Declaration

```
virtual void Send(
    Destination dest,
    Message message,
    MessageDeliveryMode deliveryMode,
    int priority,
    long timeToLive );

virtual void Send(
    Message message,
    MessageDeliveryMode deliveryMode,
    int priority,
    long timeToLive );

virtual void Send(
    Destination dest,
    Message message,
    int deliveryMode,
    int priority,
    long timeToLive );

virtual void Send(
    Message message,
    int deliveryMode,
    int priority,
    long timeToLive );

virtual void Send(
    Destination dest,
    Message message );

virtual void Send(
    Message message );
```

Purpose Send a message.

Parameter	Description
dest	When present, the call sends the message to this destination (queue or topic).  When absent, the call sends the message to the producer’s default destination. When the producer does not specify a default, the send call must supply this parameter.
message	Send this message object.

Parameter	Description
<code>deliveryMode</code>	<p>When present, the call sends the message with this delivery mode.</p> <p>This argument may be either an enumerated value (see <code>MessageDeliveryMode</code> on page 194) or an integer (see <code>DeliveryMode</code> on page 190). We recommend enumerated values, because they enable .NET to do stronger type checking at compile time, which can enhance program reliability.</p> <p>When absent, the call sends the message with the producer's default delivery mode.</p>
<code>priority</code>	<p>When present, the call sends the message with this priority.</p> <p>Priority affects the order in which the server delivers messages to consumers (higher values first). The JMS specification defines ten levels of priority value, from zero (lowest priority) to 9 (highest priority). The specification suggests that clients consider 0–4 as gradations of normal priority, and priorities 5–9 as gradations of expedited priority.</p> <p>When absent, the call sends the message with the producer's default priority.</p>
<code>timeToLive</code>	<p>When present, the call uses this value (in milliseconds) to compute the message expiration.</p> <ul style="list-style-type: none"> <li>• If the time-to-live is non-zero, the expiration is the sum of that time-to-live and the sending client's current time (GMT). This rule applies even within sessions with transaction semantics—the timer begins with the send call, not the commit call.</li> <li>• If the time-to-live is zero, then expiration is also zero—indicating that the message never expires.</li> </ul> <p>When absent, the call uses the producer's default value to compute expiration.</p> <p>Whenever your application uses non-zero values for message expiration or time-to-live, you must ensure that clocks are synchronized among all the host computers that send and receive messages. Synchronize clocks to a tolerance that is a very small fraction of the smallest or time-to-live.</p>

# QueueSender

Class

- Declaration

class QueueSender : MessageProducer
- Purpose

Send messages to a queue.
- Remarks

This class extends MessageProducer on page 90. It overloads more send methods, specializing the destination parameter to a queue.

Member	Description
Properties	
Queue	<div>Queue {get;}</div> <div>Each send call directs a message to a queue.</div> <div>Programs can use this property to define a default queue for messages that this producer sends. Individual sending calls can override this default value.</div> <div>Programs set this queue when creating the sender, and cannot subsequently change it.</div>

Method	Description	Page
QueueSender.Send	Send a message.	97

# QueueSender.Send

Method

**Declaration**

```

virtual void Send(
    Queue queue,
    Message message,
    MessageDeliveryMode deliveryMode,
    int priority,
    long timeToLive );

virtual void Send(
    Queue queue,
    Message message,
    int deliveryMode,
    int priority,
    long timeToLive );

virtual void Send(
    Queue queue,
    Message message );

```

**Purpose** Send a message.

**Remarks** QueueSender inherits send methods from MessageProducer, and also defines these methods, which specialize the destination parameter to a queue; see MessageProducer.Send on page 94.

Parameter	Description
queue	<p>When present, the call sends the message to this queue.</p> <p>When absent, the call sends the message to the sender's default queue. When the sender does not specify a default, the send call must supply this parameter (that is, it cannot use one of the inherited methods that omit this parameter).</p>
message	Send this message object.
deliveryMode	<p>When present, the call sends the message with this delivery mode.</p> <p>This argument may be either an enumerated value (see MessageDeliveryMode on page 194) or an integer (see DeliveryMode on page 190). We recommend enumerated values, because they enable .NET to do stronger type checking at compile time, which can enhance program reliability.</p> <p>When absent, the call sends the message with the sender's default delivery mode.</p>

Parameter	Description
priority	<p>When present, the call sends the message with this priority.</p> <p>Priority affects the order in which the server delivers messages to consumers (higher values first). The JMS specification defines ten levels of priority value, from zero (lowest priority) to 9 (highest priority). The specification suggests that clients consider 0–4 as gradations of normal priority, and priorities 5–9 as gradations of expedited priority.</p> <p>When absent, the call sends the message with the sender’s default priority.</p>
timeToLive	<p>When present, the call uses this value (in milliseconds) to compute the message expiration.</p> <ul style="list-style-type: none"><li>• If the time-to-live is non-zero, the expiration is the sum of that time-to-live and the sending client’s current time (GMT).</li><li>• If the time-to-live is zero, then expiration is also zero—indicating that the message never expires.</li></ul> <p>When absent, the call uses the sender’s default value to compute expiration.</p> <p>Whenever your application uses non-zero values for message expiration or time-to-live, you must ensure that clocks are synchronized among all the host computers that send and receive messages. Synchronize clocks to a tolerance that is a very small fraction of the smallest or time-to-live.</p>

**See Also**    `MessageProducer.Send` on page 94



# TopicPublisher

Class

Declaration	<code>class TopicPublisher : MessageProducer</code>
Purpose	Send a message to a topic.
Remarks	This class extends <code>MessageProducer</code> on page 90. It overloads more send methods, specializing the destination parameter to a topic.

Member	Description
Properties	
Topic	<div>Topic {get;}</div> <div>Each send call directs a message to a topic.</div> <div>Programs can use this property to define a default topic for messages that this publisher sends. Individual sending calls can override this default value.</div> <div>Programs set this topic when creating the publisher, and cannot subsequently change it.</div>

Method	Description	Page
<code>TopicPublisher.Publish</code>	Publish a message to a topic.	100

# TopicPublisher.Publish

Method

Declaration

```
virtual void Publish(  
    Topic topic,  
    Message message,  
    MessageDeliveryMode deliveryMode,  
    int priority,  
    long timeToLive );  
  
virtual void Publish(  
    Message message,  
    MessageDeliveryMode deliveryMode,  
    int priority,  
    long timeToLive );  
  
virtual void Publish(  
    Topic topic,  
    Message message,  
    int deliveryMode,  
    int priority,  
    long timeToLive );  
  
virtual void Publish(  
    Message message,  
    int deliveryMode,  
    int priority,  
    long timeToLive );  
  
virtual void Publish(  
    Topic topic,  
    Message message );  
  
virtual void Publish(  
    Message message );
```

**Purpose** Publish a message to a topic.

**Remarks** These methods are parallel to the send methods that TopicPublisher inherits from MessageProducer, and they accomplish the same goal—sending messages.

Parameter	Description
topic	When present, the call sends the message to this topic.  When absent, the call sends the message to the publisher’s default topic. When the publisher does not specify a default, the publish call must supply this parameter.
message	Publish this message.

Parameter	Description
<code>deliveryMode</code>	<p>When present, the call sends the message with this delivery mode.</p> <p>This argument may be either an enumerated value (see <code>MessageDeliveryMode</code> on page 194) or an integer (see <code>DeliveryMode</code> on page 190). We recommend enumerated values, because they enable .NET to do stronger type checking at compile time, which can enhance program reliability.</p> <p>When absent, the call sends the message with the publisher's default delivery mode.</p>
<code>priority</code>	<p>When present, the call sends the message with this priority.</p> <p>Priority affects the order in which the server delivers messages to consumers (higher values first). The JMS specification defines ten levels of priority value, from zero (lowest priority) to 9 (highest priority). The specification suggests that clients consider 0–4 as gradations of normal priority, and priorities 5–9 as gradations of expedited priority.</p> <p>When absent, the call sends the message with the publisher's default priority.</p>
<code>timeToLive</code>	<p>When present, the call uses this value (in milliseconds) to compute the message expiration.</p> <ul style="list-style-type: none"> <li>• If the time-to-live is non-zero, the expiration is the sum of that time-to-live and the sending client's current time (GMT).</li> <li>• If the time-to-live is zero, then expiration is also zero—indicating that the message never expires.</li> </ul> <p>When absent, the call uses the publisher's default value to compute expiration.</p> <p>Whenever your application uses non-zero values for message expiration or time-to-live, you must ensure that clocks are synchronized among all the host computers that send and receive messages. Synchronize clocks to a tolerance that is a very small fraction of the smallest or time-to-live.</p>

**See Also** `MessageProducer.Send` on page 94



## Chapter 7 Requestor

Requestors implement convenience methods for request-reply semantics. They send messages (called *requests*) and wait for *reply* messages in response.

### Topics

---

- *QueueRequestor*, page 104
- *TopicRequestor*, page 108

# QueueRequestor

Class

- Declaration

class QueueRequestor
- Purpose

Encapsulate request-reply semantics, sending requests to a queue.
- Remarks

We recommend that programs follow these steps:

1. Create a QueueSession, and use it to create a Queue for requests and replies.

2. Create a QueueRequestor, using the queue session and queue as arguments.

3. Send a request and receive a reply. You may repeat this step for several request and reply pairs.

4. Close the requestor object. The Close method also closes the requestor's session as a side effect.

Method	Description	Page
QueueRequestor	Create a queue requestor.	105
QueueRequestor.Close	Close a queue requestor.	106
QueueRequestor.Request	Send a request message; wait for a reply and return it.	107

See Also

Queue on page 66

QueueSession on page 170

# QueueRequestor

## Constructor

Declaration	<code>QueueRequestor ( QueueSession session, Queue queue );</code>
Purpose	Create a queue requestor.

Parameter	Description
session	<p>The requestor operates within this queue session.</p> <p>This session must not use transaction semantics. Its delivery mode must be either <code>AutoAcknowledge</code> or <code>DupsOkAcknowledge</code>.</p> <p>The <code>Close</code> method also closes this session as a side effect.</p>
queue	<p>The requestor sends request messages to this queue, and waits for replies on the same queue.</p> <p>You must create this queue using the queue session you supply as the first argument.</p>

## QueueRequestor.Close

---

Method

Declaration	void Close ();
Purpose	Close a queue requestor.
Remarks	This method also closes the requestor's session as a side effect.



# QueueRequestor.Request

Method

- Declaration

Message Request(  
    Message message );
- Purpose

Send a request message; wait for a reply and return it.
- Remarks

The requestor receives only the first reply. It discards other replies that arrive subsequently.

Parameter	Description
message	Send this request message.

# TopicRequestor

Class

- Declaration

class TopicRequestor
- Purpose

Encapsulate request-reply semantics, sending requests to a topic.
- Remarks

We recommend that programs follow these steps:

1. Create a TopicSession, and use it to create a Topic for requests and replies.

2. Create a TopicRequestor, using the topic session and topic as arguments.

3. Send a request and receive a reply. You may repeat this step for several request and reply pairs.

4. Close the requestor object. The Close method also closes the requestor's session as a side effect.

Method	Description	Page
TopicRequestor	Create a topic requestor.	109
TopicRequestor.Close	Close a topic requestor.	110
TopicRequestor.Request	Send a request message; wait for a reply and return it.	111

See Also

Topic on page 72

TopicSession on page 171

# TopicRequestor

Constructor

**Declaration**

`TopicRequestor (
 TopicSession session,
 Topic topic );`

**Purpose**

Create a topic requestor.

Parameter	Description
session	<p>The requestor operates within this topic session.</p> <p>This session must not use transaction semantics. Its delivery mode must be either <code>AutoAcknowledge</code> or <code>DupsOkAcknowledge</code>.</p> <p>The <code>Close</code> method also closes this session as a side effect.</p>
topic	<p>The requestor sends request messages to this topic, and waits for replies on the same topic.</p> <p>You must create this topic using the topic session you supply as the first argument.</p>

# TopicRequestor.Close

---

Method

Declaration	void Close ();
Purpose	Close a topic requestor.
Remarks	This method also closes the requestor's session as a side effect.

# TopicRequestor.Request

Method

- Declaration

Message Request(  
    Message message );
- Purpose

Send a request message; wait for a reply and return it.
- Remarks

The requestor receives only the first reply. It discards other replies that arrive subsequently.

Parameter	Description
message	Send this request message.



## Chapter 8      **Connection**

Connection objects represent a client program's network connection to the server.

### Topics

---

- *Connection*, page 114
- *ConnectionMetaData*, page 121
- *QueueConnection*, page 122
- *TopicConnection*, page 124
- *EMSEExceptionHandler*, page 126
- *EMSEExceptionEventArgs*, page 127
- *IExceptionListener*, page 129

# Connection

Class

Declaration	<code>class Connection</code>
Purpose	Encapsulate a server connection.
Remarks	<p>When a program first opens a connection, the connection is <i>stopped</i>—that is, it does not deliver inbound messages. To begin the flow of inbound messages, the program must explicitly call the <code>Start</code> method. (Outbound messages flow even before calling <code>Start</code>.)</p> <p>The EMS .NET API does <i>not</i> support the optional methods <code>createConnectionConsumer</code> and <code>createDurableConnectionConsumer</code>.</p>
Asynchronous Exceptions	<p>When a program uses a connection to send messages, the send calls can detect problems with the connection, and notify the client program (synchronously) by throwing exceptions.</p> <p>However, when a program uses a connection only to receive messages, the client cannot catch such exceptions. Instead, programs can handle such exceptions asynchronously in one of two idioms. Programmers may select either idiom—but not both (which would cause duplicate exception processing, with undefined behavior).</p> <ul style="list-style-type: none"><li>• <code>ExceptionHandler</code> events detect this type of problem in a .NET programming idiom.</li><li>• In contrast, the <code>ExceptionListener</code> property mimics the way in which JMS provides similar functionality in a Java programming idiom.</li></ul>

(Sheet 1 of 3)

Member	Description
Events	
<code>ExceptionHandler</code>	<p><code>EMSEExceptionHandler</code></p> <p>The client library raises an event if it detects a problem with the connection. The program implements a handler delegate to processes it asynchronously, and registers the delegate here. See <code>Asynchronous Exceptions</code>, above. See <code>EMSEExceptionHandler</code> on page 126.</p>



(Sheet 2 of 3)

Member	Description
<b>Properties</b>	
ActiveURL	<p><code>string {get;}</code></p> <p>This property holds the URL of the server at the other endpoint of the connection. When the connection interacts with several servers in a fault-tolerant arrangement, this property indicates the current active server.</p>
ClientID	<p><code>string {get; set;}</code></p> <p>This property holds the unique client ID of the connection.</p> <p>Client IDs partition the namespace of durable subscribers; see <code>Session.CreateDurableSubscriber</code> on page 153.</p> <p>Administrators can configure <code>ConnectionFactory</code> objects to assign client IDs to new connections. Alternatively, administrators can allow client programs to assign their own IDs. If the factory does not assign an ID, the program may set this property. However, it is illegal to overwrite an existing client ID value, and or to set this property after using the connection in any way (for example, after creating a session, or starting the connection); attempting to set this property in these situations results in <code>IllegalStateException</code>.</p>
ExceptionListener	<p><code>IExceptionListener {get; set;}</code></p> <p>This is an alternate pathway for alerting a client program of connection problems. The program implements the exception listener interface, and registers an exception listener object by setting this property. When the client library detects a connection problem, it calls the listener's <code>onException</code> method with an exception argument that details the problem.</p> <p>See Asynchronous Exceptions, above. See <code>IExceptionListener</code> on page 129.</p>
IsClosed	<p><code>bool {get;}</code></p> <p>This property is <code>true</code> if the connection has been closed; otherwise <code>false</code>.</p>
IsSecure	<p><code>bool {get;}</code></p> <p>This property is <code>true</code> if the connection communicates with a secure protocol; otherwise <code>false</code>.</p>

(Sheet 3 of 3)

Member	Description
MetaData	ConnectionMetaData {get;}  Programs can get the connection’s metadata object.

Method	Description	Page
Connection.Close	Close the connection; reclaim resources.	117
Connection.CreateSession	Create a session object.	118
Connection.Start	Start delivering inbound messages.	119
Connection.Stop	Stop delivering inbound messages.	120

**Subclasses**

QueueConnection on page 122  
TopicConnection on page 124

## Connection.Close

### Method

<b>Declaration</b>	<code>virtual void Close();</code>
<b>Purpose</b>	Close the connection; reclaim resources.
<b>Remarks</b>	<p>Closing the connection is sufficient to reclaim all of its resources; you need not separately close the sessions, producers, and consumers associated with the connection.</p> <p>Closing a connection deletes all temporary destinations associated with the connection.</p>
<b>Blocking</b>	If any message listener or receive call associated with the connection is processing a message when the program calls this method, all facilities of the connection and its sessions remain available to those listeners until they return. In the meantime, this method blocks until that processing completes—that is, until all message listeners and receive calls have returned.
<b>Acknowledge</b>	Closing a connection does <i>not</i> force acknowledgment in client-acknowledged sessions. When the program still has a message that it received from a connection that has since closed, its <code>Message.Acknowledge</code> method throws <code>InvalidOperationException</code> .
<b>Transactions</b>	Closing a connection rolls back all open transactions in all sessions associated with the connection.
<b>See Also</b>	<p><code>Message.Acknowledge</code> on page 27</p> <p><code>MessageConsumer</code> on page 76</p> <p><code>MessageProducer</code> on page 90</p> <p><code>Destination</code> on page 65</p> <p><code>Session</code> on page 142</p> <p><code>IMessageListener</code> on page 86</p> <p><code>InvalidOperationException</code> on page 236</p>

# Connection.CreateSession

Method

**Declaration**

```
virtual Session CreateSession(  
    bool transacted,  
    SessionMode acknowledgeMode );  
  
virtual Session CreateSession(  
    bool transacted,  
    int acknowledgeMode );
```

**Purpose** Create a session object.

**Remarks** The new session uses the connection for all server communications.

Parameter	Description
transacted	When true, the new session has transaction semantics.  When false, it has non-transaction semantics.
acknowledgeMode	This parameter determines the acknowledge mode of the session.  Supply a value enumerated by the members of SessionMode.  For backward compatibility, you may also supply an integer value from the members of Session.

**See Also** [Message.Acknowledge](#) on page 27  
[Session](#) on page 142  
[Acknowledge Modes](#) on page 146  
[SessionMode](#) on page 168

## Connection.Start

---

*Method*

<b>Declaration</b>	<code>virtual void Start();</code>
<b>Purpose</b>	Start delivering inbound messages.
<b>Remarks</b>	<p>When a connection is created, it is stopped. It does not deliver inbound messages until the program calls this method to explicitly start it.</p> <p>If the connection is not stopped, this call has no effect.</p> <p>Outbound messages flow even before calling <code>Start</code>.</p>
<b>See Also</b>	<code>Connection.Stop</code> on page 120

# Connection.Stop

Method

Declaration	<code>virtual void Stop();</code>
Purpose	Stop delivering inbound messages.
Remarks	<p>This call temporarily stops the connection from delivering inbound messages. A program can restart delivery by calling <code>Connection.Start</code>.</p> <p>When a connection is created, it is stopped. It does not deliver inbound messages until the program calls this method to explicitly start it.</p> <p>If the connection is already stopped, this call has no effect.</p>
Effect	<p>When this call returns, the connection has stopped delivery to all consumers associated with the connection:</p> <ul style="list-style-type: none"><li>• Messages do not arrive to trigger asynchronous message handler events, nor message listeners.</li><li>• Synchronous receive methods block. If their timeout intervals expire, they return null.</li></ul>
Blocking	<p>If any message listener or receive call associated with the connection is processing a message when the program calls this method, all facilities of the connection and its sessions remain available to those listeners until they return. In the meantime, this method blocks until that processing completes—that is, until all message listeners and receive calls have returned.</p> <p>However, the stopped connection prevents the client program from processing any new messages.</p>
Sending	A stopped connection can still send outbound messages.
See Also	<code>Connection.Start</code> on page 119

# ConnectionMetaData

*Class*

<b>Declaration</b>	<code>class ConnectionMetaData</code>
<b>Purpose</b>	Information about the provider.
<b>Remarks</b>	Programs can retrieve this object from a connection; see <code>MetaData</code> on page 116.

Member	Description
<b>Properties</b>	
<code>JMSXPropertyNames</code>	<code>System.Collections.IEnumerator {get;}</code> Enumerates the JMS message properties; see <code>JMS Properties</code> on page 19.
<code>MajorVersion</code>	<code>int {get;}</code> Major version number of the JMS specification that the provider supports.
<code>MinorVersion</code>	<code>int {get;}</code> Minor version number of the JMS specification that the provider supports.
<code>Version</code>	<code>string {get;}</code> Version number of the JMS specification that the provider supports.
<code>ProviderMajorVersion</code>	<code>int {get;}</code> Major version number of the provider (EMS).
<code>ProviderMinorVersion</code>	<code>int {get;}</code> Minor version number of the provider (EMS).
<code>ProviderVersion</code>	<code>string {get;}</code> Version number of the provider (EMS).
<code>ProviderName</code>	<code>string {get;}</code> Vendor name of the provider.

# QueueConnection

Class

- Declaration

class QueueConnection : Connection
- Purpose

Backward compatibility. Connection restricted to queue operations.
- Remarks

This class supports existing programs that use it.

For new programs, we recommend using the more general class, Connection on page 114, instead.

Method	Description	Page
QueueConnection.CreateQueueSession	Backward compatibility. Create a queue session object.	123



# QueueConnection.CreateQueueSession

Method

Declaration

```
virtual QueueSession CreateQueueSession(
    bool transacted,
    SessionMode acknowledgeMode );

virtual QueueSession CreateQueueSession(
    bool transacted,
    int acknowledgeMode );
```

Purpose

Backward compatibility. Create a queue session object.

Remarks

The new queue session uses the connection for all server communications.

Parameter	Description
transacted	When true, the new session has transaction semantics.  When false, it has non-transaction semantics.
acknowledgeMode	This parameter determines the acknowledge mode of the session.  Supply a value enumerated by the members of SessionMode.  For backward compatibility, you may also supply an integer value from the members of Session.

See Also

Message.Acknowledge on page 27  
QueueSession on page 170  
Acknowledge Modes on page 146  
SessionMode on page 168

# TopicConnection

Class

- Declaration

class TopicConnection : Connection
- Purpose

Backward compatibility. Connection restricted to topic operations.
- Remarks

This class supports existing programs that use it.

For new programs, we recommend using the more general class, Connection on page 114, instead.

Method	Description	Page
TopicConnection.CreateTopicSession	Backward compatibility. Create a topic session object.	125

# TopicConnection.CreateTopicSession

Method

<b>Declaration</b>	<pre>virtual TopicSession CreateTopicSession(     bool transacted,     SessionMode acknowledgeMode );  virtual TopicSession CreateTopicSession(     bool transacted,     int acknowledgeMode );</pre>
<b>Purpose</b>	Backward compatibility. Create a topic session object.
<b>Remarks</b>	The new topic session uses the connection for all server communications.

Parameter	Description
transacted	<p>When true, the new session has transaction semantics.</p> <p>When false, it has non-transaction semantics.</p>
acknowledgeMode	<p>This parameter determines the acknowledge mode of the session.</p> <p>Supply a value enumerated by the members of SessionMode.</p> <p>For backward compatibility, you may also supply an integer value from the members of Session.</p>

<b>See Also</b>	<p>Message.Acknowledge on page 27</p> <p>TopicSession on page 171</p> <p>Acknowledge Modes on page 146</p> <p>SessionMode on page 168</p>
-----------------	---

# EMSEExceptionHandler

Delegate

**Declaration**      `delegate void EMSEExceptionHandler(  
                         object sender,  
                         EMSExceptionEventArgs args );`

**Purpose**            Asynchronously detect problems with connections.

**Remarks**        When a program uses a connection to send messages, the send calls can detect problems with the connection, and notify the client program by throwing exceptions. However, when a program uses a connection only to receive messages, the client cannot catch such exceptions.

This delegate provides an alternate pathway for alerting a client program of connection problems. The program implements this delegate, and registers it with the connection. When the client library detects a connection problem, it raises an event. This delegate processes the event, which contains an exception that details the problem.

EMSEExceptionHandler detects this type of problem in a .NET programming idiom. In contrast, IExceptionListener mimics the way in which JMS provides similar functionality in a Java programming idiom. Programmers may select either idiom—but not both (which would cause duplicate exception processing, with undefined behavior).

Parameter	Description
sender	The problematic connection object.
args	The event, which contains the exception object.

Example 2    *Exception Event Handler*

```
...  
connection.ExceptionHandler += new EMSEExceptionHandler(handleEx);  
...  
private void handleEx(object sender, EMSExceptionEventArgs arg)  
{  
    EMSException e = arg.Exception;  
    Console.WriteLine("Exception: " + e.Message);  
}  
...
```

**See Also**        Connection on page 114  
                     EMSEExceptionHandler on page 126  
                     EMSExceptionEventArgs on page 127

# EMSExceptionEventArgs

Class

- Declaration**

`class EMSMessageEventArgs : EventArgs`
- Purpose**

Present a connection problem as a .NET event.
- Remarks**

EMSExceptionHandler delegates receive this object as an argument.

Member	Description
Properties	
Exception	<div>EMSException {get;}</div> <div>Programs can get the exception that details the problem.</div>

Method	Description	Page
EMSMessageEventArgs	Constructor.	85

**See Also**    EMSExceptionHandler on page 126

# EMSExceptionEventArgs

## Constructor

**Declaration**

```
EMSExceptionEventArgs(  
    EMSException emse );
```

**Purpose**

Create an exception event.

Parameter	Description
emse	The new event encapsulates this exception, and signals its arrival.

# IExceptionListener

## Interface

**Declaration**     `interface IExceptionListener`

**Purpose**            Asynchronously detect problems with connections.

**Remarks**        When a program uses a connection to send messages, the send calls can detect problems with the connection, and notify the client program by throwing exceptions. However, when a program uses a connection only to receive messages, the client cannot catch such exceptions.

This interface provides an alternate pathway for alerting a client program of connection problems. The program implements this interface, and registers an exception listener with the connection object. When the client library detects a connection problem, it calls the listener's `onException` method with an exception argument that details the problem.

`IExceptionListener` mimics the way in which JMS detects this type of problem in a Java programming idiom. In contrast, `EMSEExceptionHandler` provides similar functionality in a .NET idiom. Programmers may select either idiom—but not both (which would cause duplicate exception processing, with undefined behavior).

Method	Description	Page
<code>IExceptionListener.OnException</code>	Handle connection exceptions asynchronously.	130

**See Also**        Connection on page 114  
                 `EMSEExceptionHandler` on page 126

# IExceptionHandler.OnException

Method

Declaration

Purpose

```
void OnException(  
    EMSException exception );
```

Handle connection exceptions asynchronously.

Parameter	Description
exception	Handle this exception.



## Chapter 9 **Connection Factory**

Connection factories let administrators preconfigure client connections to the EMS server.

### Topics

---

- *ConnectionFactory, page 132*
- *QueueConnectionFactory, page 137*
- *TopicConnectionFactory, page 139*

# ConnectionFactory

Class

Declaration	<code>class ConnectionFactory</code>
Purpose	Administered object for creating server connections.
Remarks	<p>Connection factories are administered objects. They support concurrent use.</p> <p>Administrators define connection factories in a repository. Each connection factory has administrative parameters that guide the creation of server connections. Usage follows either of two models:</p>
EMS Server	<p>You can use the EMS server as a name service provider—one <code>tibemsd</code> process provides both the name repository and the message service. Administrators define factories in the name repository. Client programs create connection factory objects with the URL of the repository, and call the <code>ConnectionFactory.CreateConnection</code> method. This method automatically accesses the corresponding factory in the repository, and uses it to create a connection to the message service.</p>
Separate JNDI Repository	<p>Administrators define factories in a JNDI repository. Client programs call <code>LookupContext.Lookup</code> to retrieve factories, and use them to create connections to the server.</p>

Member	Description
Properties	
Metric	<p><code>FactoryLoadBalanceMetric Metric {get; set;}</code></p> <p>When the connection factory balances the client load among several servers, it uses this metric to determine the least loaded server, so the connection factory can create a connection to it. For values, see <code>FactoryLoadBalanceMetric</code> on page 136.</p>

Method	Description	Page
<code>ConnectionFactory</code>	Constructor.	134
<code>ConnectionFactory.CreateConnection</code>	Create a connection object.	135

<b>Administered Objects</b>	Administered objects let administrators configure EMS behavior at the enterprise level. Administrators define these objects, and client programs use them. This arrangement relieves program developers and end users of the responsibility for correct configuration.
<b>Subclasses</b>	QueueConnectionFactory on page 137 TopicConnectionFactory on page 139
<b>See Also</b>	LookupContext on page 182

# ConnectionFactory

## Constructor

**Declaration**

```
ConnectionFactory(  
    string serverUrl,  
    string clientId,  
    Hashtable properties );  
  
ConnectionFactory(  
    string serverUrl,  
    string clientId );  
  
ConnectionFactory(  
    string serverUrl );  
  
ConnectionFactory();
```

**Purpose**

Create a connection factory.

**Remarks**

When administrators define factories in the EMS server, these constructors automatically access the corresponding objects in the repository.

Parameter	Description
serverUrl	The constructor contacts the EMS server at this URL, to access a factory.
clientId	A client ID string lets the server associate a client-specific factory with each client program. When present, the server supplies that factory to the client. If a factory does not yet exist for the client, the server creates one, and stores it for future use by that specific client.
properties	When present, these properties govern the behavior of the connection objects that a client-specific factory creates. For a list of properties, see Connection-Related Fields (Constants) on page 195.

**Reconnect and Fault Tolerance**

To enable reconnection behavior and fault tolerance, the `serverURL` parameter must be a comma-separated list of two or more URLs. In a situation with only one server, you may supply two copies of that server’s URL to enable client reconnection (for example, `tcp://localhost:7222,tcp://localhost:7222`).

**See Also**

LookupContext on page 182

# ConnectionFactory.CreateConnection

Method

Declaration

```
Connection CreateConnection(  
    string userName,  
    string password );  
  
Connection CreateConnection();
```

Purpose

Create a connection object.

When the identity parameters are absent, the connection object presents a default user identity. If the server configuration permits that user, then the call succeeds.

Parameter	Description
userName	When present, the connection object presents this user identity to the server.
password	When present, the connection object authenticates the user identity with this password.

See Also

Connection on page 114

# FactoryLoadBalanceMetric

Class

- Declaration

enum FactoryLoadBalanceMetric
- Purpose

Define enumerated load balancing constants.
- Remarks

When a connection factory balances the client load among several servers, it uses this metric to determine the least loaded server, so the connection factory can create a connection to it.

Member	Description
Fields	
None	Indicates absence of any load balancing metric.
Connections	The connection factory balances the connection load among several servers by creating a connection to the server with the fewest number of connections.
ByteRate	The connection factory balances the connection load among several servers by creating a connection to the server with the lowest total byte rate (input and output).
See Also	ConnectionFactory on page 132

# QueueConnectionFactory

Class

- Declaration**

class QueueConnectionFactory : ConnectionFactory
- Purpose**

Backward compatibility. Administered object for creating queue connections.
- Remarks**

This class supports existing programs that use it.

For new programs, we recommend using the more general class, ConnectionFactory on page 132, instead.

Method	Description	Page
QueueConnectionFactory	Constructor.  Same method signatures as the constructors for ConnectionFactory on page 134.	—
QueueConnectionFactory.CreateQueueConnection	Create a queue connection object.	138

## QueueConnectionFactory.CreateQueueConnection

Method

- Declaration**

```
QueueConnection CreateQueueConnection(  
    string userName,  
    string password );  
  
QueueConnection CreateQueueConnection();
```
- Purpose** Create a queue connection object.
- Remarks** When the identity parameters are absent, the connection object presents a default user identity. If the server configuration permits that user, then the call succeeds.

Parameter	Description
userName	When present, the connection object presents this user identity to the server.
password	When present, the connection object authenticates the user identity with this password.

**See Also** QueueConnection on page 122



# TopicConnectionFactory

Class

- Declaration**

class TopicConnectionFactory : ConnectionFactory
- Purpose**

Backward compatibility. Administered object for creating topic connections.
- Remarks**

This class supports existing programs that use it.

For new programs, we recommend using the more general class, ConnectionFactory on page 132, instead.

Method	Description	Page
TopicConnectionFactory	Constructor.  Same method signatures as the constructors for ConnectionFactory on page 134.	—
TopicConnectionFactory.CreateTopicConnection	Create a topic connection object.	140

# TopicConnectionFactory.CreateTopicConnection

Method

- Declaration

```
TopicConnection CreateTopicConnection(  
    string userName,  
    string password );  
  
TopicConnection CreateTopicConnection();
```
- Purpose

Create a topic connection object.
- Remarks

When the identity parameters are absent, the connection object presents a default user identity. If the server configuration permits that user, then the call succeeds.

Parameter	Description
userName	When present, the connection object presents this user identity to the server.
password	When present, the connection object authenticates the user identity with this password.

**See Also**

TopicConnection on page 124

## Chapter 10    **Session**

A session is a single-threaded context for producing and consuming messages.

### Topics

---

- *Session, page 142*
- *SessionMode, page 168*
- *QueueSession, page 170*
- *TopicSession, page 171*

# Session

Class

Declaration	<code>class Session</code>
Purpose	Main organizing context for message activity.
Remarks	<div>Sessions combine several roles:<ul style="list-style-type: none"><li>• Factory for message producers and consumers</li><li>• Factory for message objects</li><li>• Factory for temporary destinations</li><li>• Factory for dynamic destinations</li><li>• Factory for queue browsers</li><li>• Serializer for inbound and outbound messages</li><li>• Serializer for asynchronous message events (or message listeners) of its consumer objects</li><li>• Cache for inbound messages (until the program acknowledges them).</li><li>• Transaction support (when enabled).</li></ul></div>
Single Thread	The JMS specification restricts programs to use each session within a single thread.
Associated Objects	The same single-thread restriction applies to objects associated with a session—namely, messages, message consumers, durable subscribers, message producers, queue browsers, and temporary destinations (however, static and dynamic destinations are exempt from this restriction).
Corollary	One consequence of this rule is that all the consumers of a session must deliver messages in the same mode—either synchronously or asynchronously.
Asynchronous	In asynchronous delivery, the program registers message handler events or message listeners with the session’s consumer objects. An internal dispatcher thread delivers messages to those event handlers or listeners (in all the session’s consumer objects). No other thread may use the session (nor objects created by the session).

**Synchronous** In synchronous delivery, the program explicitly begins a thread for the session. That thread processes inbound messages and produces outbound messages, serializing this activity among the session’s producers and consumers. Methods that request the next message (such as `MessageConsumer.Receive`) can organize the thread’s activity.

**Close** The only exception to the rule restricting session calls to a single thread is the method `Session.Close`; programs can call `Close` from any thread at any time.


- Transactions** A session has either transaction or non-transaction semantics. When a program specifies transaction semantics, the session object cooperates with the server, and all messages that flow through the session become part of a transaction.
- When the program calls `Session.Commit`, the session acknowledges all inbound messages in the current transaction, and the server delivers all outbound messages in the current transaction to their destinations.
  - If the program calls `Session.Rollback`, the session recovers all inbound messages in the current transaction (so the program can consume them in a new transaction), and the server destroys all outbound messages in the current transaction.

After these actions, both `Commit` and `Rollback` immediately begin a new transaction.

(Sheet 1 of 2)

Member	Description
Properties	
<code>AcknowledgeMode</code>	<code>int {get;}</code>  This mode governs message acknowledgement and redelivery for consumers associated with the session. For values, see <a href="#">Acknowledge Modes</a> on page 146.  This property is irrelevant when <code>IsTransacted</code> is <code>true</code> .
<code>SessionAcknowledgeMode</code>	<code>SessionMode {get;}</code>  This parallel property accesses the same information using enumerated values (instead of ordinary integers). We recommend it over the ordinary integer-valued accessor, because it enables .NET to do stronger type checking at compile time, which can enhance program reliability. For values, see <a href="#">SessionMode</a> on page 168.

(Sheet 2 of 2)

Member	Description
Connection	<code>Connection {get;}</code> The session is associated with this connection.
IsClosed	<code>bool {get;}</code> When true, the session has been closed. When false, the session is valid.
MessageListener	<code>IMessageListener {get; set;}</code>  <b>Obsolete</b> This property is deprecated; use the property of <code>MessageConsumer</code> with same name— <code>MessageListener</code> on page 76, or <code>MessageHandler</code> on page 76.
SessID	<code>long {get;}</code> Session ID.
IsTransacted	<code>bool {get;}</code> When true, the session has transaction semantics, and <code>AcknowledgeMode</code> is irrelevant. When false, it has non-transaction semantics.

(Sheet 1 of 2)

Method	Description	Page
<b>Messages</b>		
<code>Session.CreateBytesMessage</code>	Create a byte array message.	151
<code>Session.CreateMapMessage</code>	Create a map message.	155
<code>Session.CreateObjectMessage</code>	Create an object message.	156
<code>Session.CreateStreamMessage</code>	Create a stream message.	159
<code>Session.CreateTextMessage</code>	Create a text message.	162

(Sheet 2 of 2)

Method	Description	Page
<b>Destinations</b>		
<code>Session.CreateBrowser</code>	Create a queue browser.	150
<code>Session.CreateTemporaryQueue</code>	Create a temporary queue.	160
<code>Session.CreateTemporaryTopic</code>	Create a temporary topic.	161
<code>Session.CreateQueue</code>	Create a queue.	158
<code>Session.CreateTopic</code>	Create a topic.	163
<b>Consumers &amp; Producers</b>		
<code>Session.CreateConsumer</code>	Create a message consumer.	152
<code>Session.CreateDurableSubscriber</code>	Create a durable topic subscriber.	153
<code>Session.CreateProducer</code>	Create a message producer.	157
<code>Session.Unsubscribe</code>	Unsubscribe a durable topic subscription.	167
<b>Transactions</b>		
<code>Session.Commit</code>	Commit the open transaction.	149
<code>Session.Rollback</code>	Roll back messages in the current transaction.	165
<b>Other</b>		
<code>Session.Close</code>	Close a session; reclaim resources.	148
<code>Session.Recover</code>	Recover from undetermined state during message processing.	164
<code>Session.Run</code>	Obsolete. Do not call.	166

(Sheet 1 of 2)

Field	Description
SESSION_TRANSACTED	<div>int</div> <div>The <code>IsTransacted</code> property has this value (<code>true</code>) if the session uses transaction semantics.</div>
Acknowledge Modes	
AUTO_ACKNOWLEDGE	<div>int</div> <div>In this mode, the session automatically acknowledges a message when message processing is finished—that is, in either of these methods returns successfully:</div> <div><ul style="list-style-type: none"><li>• synchronous <code>Receive</code> call</li><li>• asynchronous listener handler</li></ul></div>
CLIENT_ACKNOWLEDGE	<div>int</div> <div>In this mode, the client program acknowledges receipt by calling <code>Message.Acknowledge</code> on page 27. Each call acknowledges all messages received so far.</div>
DUPS_OK_ACKNOWLEDGE	<div>int</div> <div>As with <code>AUTO_ACKNOWLEDGE</code>, the session automatically acknowledges messages. However, it may do so lazily. <i>Lazy</i> means that the provider client library can delay transferring the acknowledgement to the server until a convenient time; meanwhile the server might redeliver the message. <i>Lazy</i> acknowledgement can reduce session overhead.</div>
EXPLICIT_CLIENT_ACKNOWLEDGE	<div>int</div> <div>As with <code>CLIENT_ACKNOWLEDGE</code>, the client program acknowledges receipt by calling <code>Message.Acknowledge</code> on page 27. However, each call acknowledges <i>only</i> the individual message. The client may acknowledge messages in any order.</div> <div>This mode and behavior are proprietary extensions, specific to TIBCO EMS.</div>



(Sheet 2 of 2)

Field	Description
EXPLICIT_CLIENT_DUPS_OK_ACKNOWLEDGE	<p>int</p> <p>In EXPLICIT_CLIENT_DUPS_OK_ACKNOWLEDGE mode, the client program lazily acknowledges <i>only</i> the individual message, by calling <code>Message.Acknowledge</code> on page 27. The client may acknowledge messages in any order.</p> <p><i>Lazy</i> means that the provider client library can delay transferring the acknowledgement to the server until a convenient time; meanwhile the server might redeliver the message.</p> <p>This mode and behavior are proprietary extensions, specific to TIBCO EMS.</p>
NO_ACKNOWLEDGE	<p>int</p> <p>In NO_ACKNOWLEDGE mode, messages do not require acknowledgement (which reduces message overhead). The server never redelivers messages.</p> <p>This mode is available for <i>topic</i> sessions only.</p> <p>This mode and behavior are proprietary extensions, specific to TIBCO EMS.</p>

# Session.Close

---

Method

Declaration	<code>void Close();</code>
Purpose	Close a session; reclaim resources.
Remarks	Closing a session automatically closes its consumers (except for durable subscribers), producers and browsers.
Blocking	If any message listener or receive call associated with the session is processing a message when the program calls this method, all facilities of the connection and its sessions remain available to those listeners until they return. In the meantime, this method blocks until that processing completes—that is, until all message listeners and receive calls have returned.
Transactions	Closing a session rolls back the open transaction in the session.

## Session.Commit

---

### Method

<b>Declaration</b>	<code>virtual void Commit();</code>
<b>Purpose</b>	Commit the open transaction.
<b>Remarks</b>	A session (with transaction semantics) always has exactly one open transaction. Message operations associated with the session become part of that transaction. This call commits all the messages within the transaction, and releases any locks. Then it opens a new transaction.
<b>Throws</b>	EMSEException on page 230 TransactionRolledBackException on page 251 IllegalStateException on page 236

# Session.CreateBrowser

Method

**Declaration**

```
QueueBrowser CreateBrowser(  
    Queue queue,  
    string messageSelector );  
  
QueueBrowser CreateBrowser(  
    Queue queue );
```

**Purpose**

Create a queue browser.

Parameter	Description
queue	Browse this queue.
messageSelector	When present, the browser presents only messages that match this selector; see Message Selectors on page 20.  When absent, null, or the empty string, the browser views all messages in the queue.

**Throws**

EMSEException on page 230  
InvalidDestinationException on page 238  
InvalidSelectorException on page 240

**See Also**

Queue on page 66  
QueueBrowser on page 174

## Session.CreateBytesMessage

---

*Method*

**Declaration**    `BytesMessage CreateBytesMessage();`

**Purpose**        Create a byte array message.

**See Also**      `BytesMessage` on page 36

# Session.CreateConsumer

## Method

Declaration

```
MessageConsumer CreateConsumer(  
    Destination dest,  
    string messageSelector,  
    bool noLocal );  
  
MessageConsumer CreateConsumer(  
    Destination dest,  
    string messageSelector );  
  
MessageConsumer CreateConsumer(  
    Destination dest );
```

Purpose

Create a message consumer.

Parameter	Description
dest	Create a consumer for this destination. The argument may be any destination (queue or topic).
messageSelector	When present, the server filters messages using this selector, so the consumer receives only matching messages; see Message Selectors on page 20.  When absent, null, or the empty string, the consumer receives messages without filtering.
noLocal	When true, the server filters messages so the consumer does not receive messages that originate locally—that is, messages sent through the same connection.  When absent or false, the consumer receives messages with local origin.

Throws

EMSEException on page 230  
IllegalStateException on page 236  
InvalidDestinationException on page 238  
InvalidSelectorException on page 240

See Also

Destination on page 65

## Session.CreateDurableSubscriber

### Method

**Declaration**

```

TopicSubscriber CreateDurableSubscriber(
    Topic topic,
    string name,
    string messageSelector,
    bool noLocal );

TopicSubscriber CreateDurableSubscriber(
    Topic topic,
    string name );

```

**Purpose** Create a durable topic subscriber.

Parameter	Description
<code>topic</code>	Create a durable subscriber for this topic (which <i>cannot</i> be a <code>TemporaryTopic</code> ).
<code>name</code>	This unique name lets the server associate the subscriber with a subscription.
<code>messageSelector</code>	When present, the server filters messages using this selector, so the subscriber receives only matching messages; see Message Selectors on page 20.  When absent, null, or the empty string, the subscriber receives messages without filtering.
<code>noLocal</code>	When <code>true</code> , the server filters messages so the subscriber does not receive messages that originate locally—that is, messages sent through the same connection.  When absent or <code>false</code> , the consumer receives messages with local origin.

**Remarks** The server associates a durable subscription with at most one subscriber object at a time. When a subscriber object exists, the subscription is *active*, and the server delivers messages to it; when no subscriber object exists, the subscription is *inactive*.

Durable subscriptions guarantee message delivery across periods during which the subscriber is inactive. The server retains unacknowledged messages until the subscriber acknowledges them, or until the messages expire.

**Subscription Continuity** Continuity across inactive periods uses two data items from the client:

- **Subscription Name** a parameter of this method
- **Client ID** an optional property of the `Connection` (used only when supplied)

The server uses one or both of these two items to match a subscriber object with its subscription. If a matching subscription exists, and it is inactive, then the server associates it with the subscriber (and the subscription becomes active). The server delivers unacknowledged messages to the subscriber.

If a matching subscription exists, but it is already active, this method throws `EMSEException`.

If a matching subscription to the topic does not yet exist, the server creates one.

**Matching Client ID**

- If the `Connection`'s client ID is non-null when a session creates a durable subscription, then only sessions of a connection with the same client ID can attach to that subscription.
- If the `Connection`'s client ID is null when a session creates a durable subscription, then any session can attach to that subscription (to receive its messages).

**Changing Topic or Selector**

Notice that the server does *not* use the topic and message selector arguments to match a subscriber to an existing subscription. As a result, client programs can *change* a subscription by altering either or both of these arguments. The effect is equivalent to deleting the existing subscription (from the server) and creating a new one (albeit with the same client ID and subscription name).

**Throws**

`EMSEException` on page 230  
`IllegalStateException` on page 236  
`InvalidDestinationException` on page 238  
`InvalidSelectorException` on page 240

**See Also**

Topic on page 72  
 Connection on page 114



## Session.CreateMapMessage

---

*Method*

**Declaration**     `MapMessage CreateMapMessage();`

**Purpose**            Create a map message.

**See Also**          `MapMessage` on page 44

# Session.CreateObjectMessage

Method

**Declaration**

```
ObjectMessage CreateObjectMessage(  
    object obj );  
  
ObjectMessage CreateObjectMessage();
```

**Purpose** Create an object message.

Parameter	Description
obj	When present, use this object as data in the new message.

**See Also** ObjectMessage on page 50

# Session.CreateProducer

Method

**Declaration**     `MessageProducer CreateProducer(  
                         Destination dest );`

**Purpose**            Create a message producer.

Parameter	Description
dest	When non-null, the producer sends messages to this destination.  When null, the client program must specify the destination for each message individually.

**Throws**            `EMSEException` on page 230  
                      `InvalidDestinationException` on page 238

**See Also**          `MessageProducer` on page 90

# Session.CreateQueue

## Method

- Declaration

Queue CreateQueue(  
    string queueName );
- Purpose

Create a queue.
- Remarks

If the named queue already exists at the server, then this method returns that queue. (That queue can be either static or dynamic.)

If the named queue does not yet exist at the server, and the server allows dynamic queues, then this method creates a dynamic queue.

Dynamic destinations are provider-specific, so programs that use them might not be portable to other providers.

Parameter	Description
queueName	Get or create the queue with this name.

**See Also** Queue on page 66

## Session.CreateStreamMessage

---

*Method*

**Declaration**     `StreamMessage CreateStreamMessage();`

**Purpose**            Create a stream message.

**See Also**         `StreamMessage` on page 52

## Session.CreateTemporaryQueue

---

Method

Declaration	TemporaryQueue CreateTemporaryQueue();
Purpose	Create a temporary queue.
Remarks	A temporary queue lasts no longer than the connection. That is, when the connection is closed or broken, the server deletes temporary queues associated with the connection.
See Also	TemporaryQueue on page 68

## Session.CreateTemporaryTopic

---

*Method*

<b>Declaration</b>	<code>TemporaryTopic CreateTemporaryTopic();</code>
<b>Purpose</b>	Create a temporary topic.
<b>Remarks</b>	A temporary topic lasts no longer than the connection. That is, when the connection is closed or broken, the server deletes temporary topic associated with the connection.
<b>See Also</b>	<code>TemporaryTopic</code> on page 70

## Session.CreateTextMessage

Method

**Declaration**

```
TextMessage CreateTextMessage(  
    string text);  
  
TextMessage CreateTextMessage();
```

**Purpose**

Create a text message.

Parameter	Description
text	When present, use this string as data in the new message.

**See Also**

TextMessage on page 59



## Session.CreateTopic

---

### Method

<b>Declaration</b>	<pre>Topic CreateTopic(     string topicName );</pre>
<b>Purpose</b>	Create a topic.
<b>Remarks</b>	<p>If the named topic already exists at the server, then this method returns that topic. (That topic can be either static or dynamic.)</p> <p>If the named topic does not yet exist at the server, and the server allows dynamic topics, then this method creates a dynamic topic.</p> <p>Dynamic destinations are provider-specific, so programs that use them might not be portable to other providers.</p>

Parameter	Description
topicName	Get or create the topic with this name.

---

**See Also**    Topic on page 72

# Session.Recover

Method

Declaration	<code>void Recover();</code>
Purpose	Recover from undetermined state during message processing.
Remarks	<p>Exceptions during message processing can sometimes leave a program in an ambiguous state. For example, some messages might be partially processed. This method lets a program return to an unambiguous state—the point within the message stream when the program last acknowledged the receipt of inbound messages. Programs can then review the messages delivered since that point (they are marked as <i>redelivered</i>), and resolve ambiguities about message processing.</p> <p>Programs can also use this method to resolve similar ambiguities after a Connection stops delivering messages, and then starts again.</p>
Operation	<p>This method requests that the server do this sequence of actions:</p> <ol style="list-style-type: none"><li>1. Stop message delivery within the session.</li><li>2. Mark as <i>redelivered</i>, any messages that the server has attempted to deliver to the session, but for which it has not received acknowledgement (that is, messages for which processing state is ambiguous).</li></ol> <p>According to the JMS specification, the server need not redeliver messages in the same order as it first delivered them.</p> <ol style="list-style-type: none"><li>3. Restart message delivery (including messages marked as <i>redelivered</i> in step 2).</li></ol> <p>When a session has transactional semantics, this method throws <code>IllegalStateException</code> (commit and rollback are more appropriate for transactions).</p>
Throws	<p><code>EMSEException</code> on page 230</p> <p><code>IllegalStateException</code> on page 236</p>

## Session.Rollback

---

### *Method*

<b>Declaration</b>	<code>virtual void Rollback();</code>
<b>Purpose</b>	Roll back messages in the current transaction.
<b>Remarks</b>	When a session does not have transactional semantics, this method throws <code>IllegalStateException</code> .
<b>Throws</b>	<code>EMSEException</code> on page 230 <code>IllegalStateException</code> on page 236

# Session.Run

---

Method



Declaration	<code>void Run();</code>
Purpose	Obsolete. Do not call.

## Session.Unsubscribe

### Method

**Declaration**     `void Unsubscribe(  
                    string name );`

**Purpose**            Unsubscribe a durable topic subscription.

**Remarks**        This method deletes the subscription from the server.

It is illegal to delete an active subscription—that is, while a `MessageConsumer` or `TopicSubscriber` exists. Attempting to do so results in `EMSException`.

It is illegal to delete a subscription while one of its messages is either unacknowledged, or uncommitted (in the current transaction). Attempting to do so results in `EMSException`.

If the session is closed, this method throws `IllegalStateException`.

Parameter	Description
name	This name lets the server locate the subscription.

**Throws**            `EMSException` on page 230  
                      `IllegalStateException` on page 236  
                      `InvalidDestinationException` on page 238

**See Also**          `MessageConsumer` on page 76  
                      `TopicSubscriber` on page 82  
                      `Topic` on page 72  
                      `Session.CreateDurableSubscriber` on page 153

# SessionMode

Enumeration

- Declaration**enum SessionMode
- Purpose**Enumerate constants associated with sessions.

(Sheet 1 of 2)

Members	Description
SessionTransacted	The IsTransacted property has this value if the session uses transaction semantics.
Acknowledge Modes	
AutoAcknowledge	<p>In this mode, the session automatically acknowledges a message when message processing is finished—that is, in either of these methods returns successfully:</p> <ul style="list-style-type: none"><li>synchronous Receive call</li><li>asynchronous listener handler</li></ul>
ClientAcknowledge	<p>In this mode, the client program acknowledges receipt by calling Message.Acknowledge on page 27. Each call acknowledges all messages received so far.</p>
DupsOkAcknowledge	<p>As with AutoAcknowledge, the session automatically acknowledges messages. However, it may do so lazily. <i>Lazy</i> means that the provider client library can delay transferring the acknowledgement to the server until a convenient time; meanwhile the server might redeliver the message. Lazy acknowledgement can reduce session overhead.</p>
ExplicitClientAcknowledge	<p>As with ClientAcknowledge, the client program acknowledges receipt by calling Message.Acknowledge on page 27. However, each call acknowledges <i>only</i> the individual message. The client may acknowledge messages in any order.</p> <p>This mode and behavior are proprietary extensions, specific to TIBCO EMS.</p>

(Sheet 2 of 2)

Members	Description
<code>ExplicitClientDupsOkAcknowledge</code>	<p>In this mode, the client program lazily acknowledges <i>only</i> the individual message, by calling <code>Message.Acknowledge</code> on page 27. The client may acknowledge messages in any order.</p> <p><i>Lazy</i> means that the provider client library can delay transferring the acknowledgement to the server until a convenient time; meanwhile the server might redeliver the message.</p> <p>This mode and behavior are proprietary extensions, specific to TIBCO EMS.</p>
<code>NoAcknowledge</code>	<p>In this mode, messages do not require acknowledgement (which reduces message overhead). The server never redelivers messages.</p> <p>This mode is available for <i>topic</i> sessions only.</p> <p>This mode and behavior are proprietary extensions, specific to TIBCO EMS.</p>

# QueueSession

---

Class

Declaration	<code>class QueueSession : Session</code>
Purpose	Session restricted to queues.
Remarks	Use this class with QueueRequestor objects.  Otherwise, when coding new programs, use the more general class, Session on page 142. Nonetheless, for backward compatibility, this class also supports existing programs that use it (rather than generic sessions).
See Also	QueueRequestor on page 104



# TopicSession

---

*Class*

**Declaration**    `class TopicSession : Session`

**Purpose**          Session restricted to topics.

**Remarks**       Use this class with `TopicRequestor` objects.  
Otherwise, when coding new programs, use the more general class, `Session` on page 142. Nonetheless, for backward compatibility, this class also supports existing programs that use it (rather than generic sessions).

**See Also**        `TopicRequestor` on page 108



## Chapter 11 Queue Browser

Queue browsers let client programs examine the messages on a queue without removing them from the queue.

### Topics

---

- *QueueBrowser*, page 174

# QueueBrowser

Class

- Declaration

class QueueBrowser : IEnumerator
- Purpose

Enumerate the messages in a queue without consuming them.
- Remarks

A browser is a dynamic enumerator of the queue (not a static snapshot). The queue is at the server, and its contents change as message arrive and consumers remove them. Meanwhile, while the browser is at the client. The method `QueueBrowser.MoveNext` asks the server for the next message after `Current`—that is, the next message that is still in the queue.

The browser can enumerate messages in a queue, or a subset filtered by a message selector.

Sessions serve as factories for queue browsers; see `Session.CreateBrowser` on page 150.

Member	Description
Properties	
Current	<div>object {get;}</div> <div>This property presents the current message in the browser’s enumeration, but accessing the property does not consume that message.</div> <div>The method <code>QueueBrowser.MoveNext</code> advances the current message.</div>
MessageSelector	<div>string {get;}</div> <div>The browser’s message selector expression filters the messages that the browser presents.</div>
Queue	<div>Queue {get;}</div> <div>The queue that this browser scans.</div>

(Sheet 1 of 2)

Method	Description	Page
<code>QueueBrowser.Close</code>	Close the browser; reclaim resources.	176

(Sheet 2 of 2)

Method	Description	Page
<code>QueueBrowser.GetEnumerator</code>	Get an enumerator of messages in the queue.	177
<code>QueueBrowser.MoveNext</code>	Advance the browser's enumeration to the next message.	178
<code>QueueBrowser.Reset</code>	Reset the browser to the location before the first message.	179

**See Also**    `Session.CreateBrowser` on page 150

# QueueBrowser.Close

---

Method

Declaration	void Close();
Purpose	Close the browser; reclaim resources.

## QueueBrowser.GetEnumerator

---

*Method*

- Declaration**    `IEnumerator GetEnumerator();`
- Purpose**        Get an enumerator of messages in the queue.
- Remarks**     This method returns the browser object—which is itself the enumerator.

# QueueBrowser.MoveNext

---

Method

Declaration	<code>bool MoveNext();</code>
Purpose	Advance the browser's enumeration to the next message.
Remarks	<p>A browser is a dynamic enumerator of the queue (not a static snapshot). The queue is at the server, and its contents change as message arrive and consumers remove them. Meanwhile, while the browser is at the client. This method asks the server for the next message after <code>Current</code>—that is, the next message that is still in the queue.</p> <p>Returns <code>true</code> if another message exists; the <code>Current</code> property subsequently presents the next message.</p> <p>Returns <code>false</code> otherwise.</p> <p>After creating a browser, programs must first call this method to examine the first message.</p>



## QueueBrowser.Reset

---

*Method*

**Declaration**    `void Reset();`

**Purpose**        Reset the browser to the location before the first message.



## Chapter 12    **Name Server Lookup**

Lookup context objects find named objects (such as connection factories and destinations) in the name repository. (The EMS server, `tibemsd`, provides the name repository service).

### Topics

---

- *LookupContext*, page 182

# LookupContext

Class

Declaration	class LookupContext
Purpose	Retrieve objects from the server’s naming directory.
Remarks	The context object establishes communication with the EMS server, authenticates the user, and submits name queries.

## Example 3 Naming Server Lookup

```
Hashtable env = new Hashtable();
env.Add(LookupContext.PROVIDER_URL,"tibjmsnaming://localhost:7222");
env.Add(LookupContext.SECURITY_PRINCIPAL", "myUserName");
env.Add(LookupContext.SECURITY_CREDENTIALS", "myPassword");
try {
    LookupContext searcher = new LookupContext(env);
    TIBCO.EMS.Queue queue = (TIBCO.EMS.Queue)searcher.Lookup("theQueueName");
    ...
} catch (NamingException)
{
    ...
}
```

(Sheet 1 of 2)

Member	Description
Properties	
Settings	System.Collections.Hashtable {get;} Programs can get a copy of the context’s current settings.
Fields	
Programs use these constants as names of context settings (in argument hashtables or individually).	
PROVIDER_URL	string URL of the naming server (EMS server).
SECURITY_CREDENTIALS	string User password of the client program.
SECURITY_PRINCIPAL	string User name of the client program.

(Sheet 2 of 2)

Member	Description
URL_LIST	<p>string</p> <p>ArrayList of URLs of naming servers (EMS servers).</p> <p>This property lets programs specify URLs as an ArrayList, rather than as a string (as with PROVIDER_URL).</p>
URL_SEPARATOR	<p>string</p> <p>Syntactic separator between URLs in the PROVIDER_URL.</p>

Method	Description	Page
LookupContext	Constructor.	184
LookupContext.AddSettings	Add or change context settings.	185
LookupContext.Lookup	Lookup an object in the naming server.	186
LookupContext.RemoveSettings	Remove a context setting.	187

# LookupContext

## Constructor

- Declaration**

```
LookupContext(  
    Hashtable prop );  
  
LookupContext();
```
- Purpose**

Create a new lookup context object.
- Remarks**

The first constructor sets properties of the new context object.  
The second constructor creates a context without property settings.

Parameter	Description
prop	Set all the name-value pairs contained in this hash table.

## LookupContext.AddSettings

Method

Declaration	<pre>virtual object AddSettings(     string propName,     object propValue );  virtual void AddSettings(     Hashtable prop );</pre>
Purpose	Add or change context settings.
Remarks	<p>The first method sets one property. If the property was previously set, the method modifies it, and returns the old value.</p> <p>The second method sets several properties with one call (like the constructor for context objects).</p>

Parameter	Description
propName	Set this single property.  For property names, see Fields on page 182.
propValue	Set the single property to this value.
prop	Set all the name-value pairs contained in this hash table.

# LookupContext.Lookup

Method

- Declaration**

```
virtual object Lookup(  
    string name );
```
- Purpose**Lookup an object in the naming server.
- Remarks**Returns the named object, if the server finds it.  
If the server does not find the object, this method throws `NamingException`.

Parameter	Description
name	Lookup this name.

**Throws** `AuthenticationException` on page 232  
`NamingException` on page 246  
`ServiceUnavailableException` on page 249



## LookupContext.RemoveSettings

---

*Method*

**Declaration**     `virtual object RemoveSettings(  
                         string propName );`

**Purpose**           Remove a context setting.

Parameter	Description
propName	Remove this property.  For property names, see Fields on page 182.

---



## Chapter 13 Utilities

This chapter presents classes and interfaces that define constants and utility methods.

### Topics

---

- *DeliveryMode*, page 190
- *IEMSSerializable*, page 191
- *MessageDeliveryMode*, page 194
- *Tibems*, page 195

# DeliveryMode

Class

- Declaration

class DeliveryMode
- Purpose

Backward compatibility. Define delivery mode constants as integers.
- Remarks

The class MessageDeliveryMode defines a parallel set of constants as .NET enumerated values (instead of ordinary integers). We recommend the enumeration over these ordinary integer values, because it enables .NET to do stronger type checking at compile time, which can enhance program reliability.

Member	Description
Fields	
NON_PERSISTENT	<div>int</div> <div>Non-persistent delivery.</div>
PERSISTENT	<div>int</div> <div>Persistent delivery.</div>
RELIABLE_DELIVERY	<div>int</div> <div>Reliable delivery mode is a TIBCO proprietary extension that offers increased performance of the message producers. See also Reliable Message Delivery on page 70 in <i>TIBCO Enterprise Message Service User's Guide</i>.</div>

# IEMSSerialziable

Interface

Declaration	interface IEMSSerialzable
Purpose	Customize serialization and deserialization of objects.
.NET Compact Framework	This interface is available only in .NET Compact Framework. Programmers can use it to allow otherwise excluded objects within an <code>ObjectMessage</code> .
Remarks	<p>When an object class implements this interface, it can be serialized within an <code>ObjectMessage</code>.</p> <p>To implement this interface for a class, define the methods listed below <i>and</i> a constructor that does not require any arguments.</p>

Method	Description	Page
IEMSSerialziable.Deserialize	Deserialize a data stream to reconstruct an object.	192
IEMSSerialziable.Serialize	Serialize an object.	193

**See Also**     `ObjectMessage` on page 50

# IEMSSerializable.Deserialize

## Method

- Declaration

```
void Deserialize(  
    System.IO.Stream stream );
```
- Purpose

Deserialize a data stream to reconstruct an object.

Parameter	Description
stream	Deserialize the data from this stream to reconstruct an object.

- Remarks

When this method is called, the stream already contains context information. That context information resides before the write position of the stream when this method is called (call it `initWritePos`). Your implementation of this method must not modify that context information, nor reset the stream's write pointer to a position before `initWritePos`.

Similarly, your implementation must not close the stream.

# IEMSSerializable.Serialize

Method

Declaration

```
void Serialize(  
    System.IO.Stream stream );
```

Purpose

Serialize an object.

Parameter	Description
stream	Serialize the object’s data to this stream.

Remarks

When this method is called, the stream already contains context information. That context information resides before the write position of the stream when this method is called (call it `initWritePos`). Your implementation of this method must not modify that context information, nor reset the stream’s write pointer to a position before `initWritePos`.

Similarly, your implementation must not close the stream.

# MessageDeliveryMode

Class

- Declaration**enum MessageDeliveryMode
- Purpose**Define enumerated delivery mode constants.
- Remarks**The class `DeliveryMode` defines a parallel set of constants as ordinary integers. However, we recommend this enumeration, because it enables .NET to do stronger type checking at compile time, which can enhance program reliability.

Member	Description
Fields	
NonPersistent	Non-persistent delivery.
Persistent	Persistent delivery.
ReliableDelivery	Reliable delivery mode is a TIBCO proprietary extension that offers increased performance of the message producers. See also Reliable Message Delivery on page 70 in <i>TIBCO Enterprise Message Service User's Guide</i> .



# Tibems

Class

<b>Declaration</b>	<code>class Tibems</code>
<b>Purpose</b>	Define constants and utility methods specific to EMS.

(Sheet 1 of 3)

Constant	Description
<b>Connection-Related Fields (Constants)</b>	
Programs can use these constants as names of settings in the hashtable argument to <code>ConnectionFactory.CreateConnection</code> . They govern the behavior of the resulting connections that the factory creates.	
<code>DEFAULT_FACTORY_PASSWORD</code>	<p>string</p> <p>Defines the name of a <code>ConnectionFactory</code> property. That property specifies a default user password for the connections that the factory creates.</p> <p>If the client does not supply a password, the connection factory object uses this password as a default when creating a connection. See also <code>ConnectionFactory.CreateConnection</code> on page 135.</p>
<code>DEFAULT_FACTORY_USERNAME</code>	<p>string</p> <p>Defines the name of a <code>ConnectionFactory</code> property. That property specifies a default username for the connections that the factory creates.</p> <p>If the client does not supply a username, the connection factory object uses this username as a default when creating a connection. See also <code>ConnectionFactory.CreateConnection</code> on page 135.</p>
<code>FACTORY_LOAD_BALANCE_METRIC</code>	<p>string</p> <p>This field defines the name of a provider-specific <code>ConnectionFactory</code> property. That property governs the assignment of client connections among a set of load-balanced servers.</p> <p>For property values, see <code>FactoryLoadBalanceMetric</code> on page 136.</p>

(Sheet 2 of 3)

Constant	Description
<b>Message-Related Fields (Constants)</b>	
Programs can use these constants as names of message properties.	
JMS_TIBCO_CM_PUBLISHER	<p>string</p> <p>Defines the name of a provider-specific message property. An imported message with that property is an RVCM message. The value of that property is the RVCM sender name. See also JMS_TIBCO_CM_PUBLISHER on page 17.</p>
JMS_TIBCO_CM_SEQUENCE	<p>string</p> <p>Defines the name of a provider-specific message property. An imported message with that property is an RVCM message. The value of that property is the RVCM sequence number. See also JMS_TIBCO_CM_SEQUENCE on page 17.</p>
JMS_TIBCO_COMPRESS	<p>string</p> <p>Defines the name of a provider-specific message property. EMS .NET does not support compression. See also JMS_TIBCO_COMPRESS on page 17.</p>
JMS_TIBCO_DISABLE_SENDER	<p>string</p> <p>Defines the name of a provider-specific message property. Programs may set that property before sending a message to request that the server omit the sender name from the message. See also JMS_TIBCO_DISABLE_SENDER on page 18.</p>
JMS_TIBCO_IMPORTED	<p>string</p> <p>Defines the name of a provider-specific message property. The server sets that property on messages it imports from external message services, such as TIBCO Rendezvous or TIBCO SmartSockets. See also JMS_TIBCO_IMPORTED on page 18.</p>
JMS_TIBCO_MSG_EXT	<p>string</p> <p>Defines the name of a provider-specific message property. When that property is set, the message can use TIBCO-specific extensions. See also JMS_TIBCO_MSG_EXT on page 18.</p>

(Sheet 3 of 3)

Constant	Description
JMS_TIBCO_MSG_TRACE	string  Defines the name of a provider-specific message property. Programs may set that property before sending a message to request trace data at significant events during the lifetime of the message. See also JMS_TIBCO_MSG_TRACE on page 18.
JMS_TIBCO_PRESERVE_UNDELIVERED	string  Defines the name of a provider-specific message property. Programs may set that property before sending a message to request that the server hold it in a special queue if the server cannot deliver it. See also JMS_TIBCO_PRESERVE_UNDELIVERED on page 18.
JMS_TIBCO_SENDER	string  Defines the name of a provider-specific message property. When a destination requests it, the server stores the username of the message producer in that property. See also JMS_TIBCO_SENDER on page 18.
JMS_TIBCO_SS_SENDER	string  Defines the name of a provider-specific message property. The server sets that property when importing a message from TIBCO SmartSockets; its value is the SmartSockets sender name. See also JMS_TIBCO_SS_SENDER on page 18.

(Sheet 1 of 3)

Method	Description	Page
Tibems.CalculateMessageSize	Returns the total size (in bytes) of a message in wire format.	200
Tibems.CreateFromBytes	Create a message from a byte array.	201
Tibems.GetAllowCloseInCallback	Determine whether client callbacks may call close methods.	202
Tibems.GetAsBytes	Copy a message into a byte array.	203

(Sheet 2 of 3)

Method	Description	Page
<code>Tibems.GetConnectAttempts</code>	Return the connection attempts setting.	204
<code>Tibems.GetEncoding</code>	Return the global character encoding for messages.	205
<code>Tibems.GetExceptionOnFTSwitch</code>	Return the fault tolerance exception setting.	206
<code>Tibems.GetMessageEncoding</code>	Return the character encoding for an individual message.	207
<code>Tibems.GetMessageSize</code> (and related methods)	Return the size of a wire format message—or its body, header or properties portions.	208
<code>Tibems.GetPingInterval</code>	Return the interval at which the client tests network connectivity.	209
<code>Tibems.GetProperty</code>	Return a property value.	210
<code>Tibems.GetReconnectAttempts</code>	Return the reconnection attempts setting.	212
<code>Tibems.GetSessionDispatcherDaemon</code>	Return the dispatcher thread setting.	213
<code>Tibems.GetSocketReceiveBufferSize</code>	Return the size of socket receive buffers.	214
<code>Tibems.GetSocketSendBufferSize</code>	Return the size of socket send buffers.	215
<code>Tibems.MakeWriteable</code>	Make a message writeable.	216
<code>Tibems.SetAllowCloseInCallback</code>	Override a JMS requirement so client callbacks may call close methods.	217
<code>Tibems.SetConnectAttempts</code>	Modify the connection attempts setting.	218
<code>Tibems.SetEncoding</code>	Set the global character encoding for messages.	219
<code>Tibems.SetExceptionOnFTSwitch</code>	Modify the fault tolerance exception setting.	220
<code>Tibems.SetMessageEncoding</code>	Set the character encoding for an individual message.	221
<code>Tibems.SetPingInterval</code>	Set the interval at which the client tests network connectivity.	222
<code>Tibems.SetProperty</code>	Modify a property value.	223

(Sheet 3 of 3)

Method	Description	Page
<code>Tibems.SetReconnectAttempts</code>	Modify the reconnection attempts setting.	225
<code>Tibems.SetSessionDispatcherDaemon</code>	Set the dispatcher thread setting.	226
<code>Tibems.SetSocketReceiveBufferSize</code>	Set the size of socket receive buffers.	227
<code>Tibems.SetSocketSendBufferSize</code>	Set the size of socket send buffers.	228

# Tibems.CalculateMessageSize

## Method

Declaration	<code>static int CalculateMessageSize(     Message msg );</code>
Purpose	Returns the total size (in bytes) of a message in wire format.
Remarks	<p>The total size includes headers, properties and body.</p> <p>This method re-measures the message, and caches the results; contrast <code>Tibems.GetMessageSize</code>.</p> <p>This method might consume process storage, and might involve disk I/O—with associated performance penalties.</p>

Parameter	Description
<code>msg</code>	Compute the size of this message.

**See Also**    `Tibems.GetMessageSize` on page 208

# Tibems.CreateFromBytes

Method

**Declaration**     `static Message CreateFromBytes(  
                         byte[] bytes );`

**Purpose**            Create a message from a byte array.

Parameter	Description
bytes	Fill the new message with this byte array.  This byte array must be the result of previously calling <code>Tibems.GetAsBytes</code> .


**Remarks**        The newly created message is read-only; to enable modification without erasing the content, call `Tibems.MakeWriteable`.

**See Also**         `Tibems.GetAsBytes` on page 203  
                      `Tibems.MakeWriteable` on page 216

# Tibems.GetAllowCloseInCallback

Method

Declaration	<code>static bool GetAllowCloseInCallback();</code>
Purpose	Determine whether client callbacks may call close methods.
Remarks	<p>According to the JMS specification, close methods (that is, <code>MessageConsumer.Close</code>, <code>Session.Close</code>, <code>Connection.Close</code>) cannot return while any message callbacks (that is, <code>EMSMessagesHandler</code>, <code>IMessageListener.OnMessage</code>) are running. As a result, a message callback must not call a close method, lest it cause a deadlock.</p> <p><code>Tibems.SetAllowCloseInCallback</code> explicitly overrides this JMS requirement, permitting callbacks to call close without deadlock (that is, embedded close calls do not wait for callbacks to return).</p>

This method replaces the deprecated method `GetAllowCallbackInClose`.

**See Also** `Tibems.SetAllowCloseInCallback` on page 217



## Tibems.GetAsBytes

---

*Method*

**Declaration**     `static byte[] GetAsBytes(  
                    Message message );`

**Purpose**            Copy a message into a byte array.

**Remarks**        The byte array includes the message headers, properties and body.

Parameter	Description
message	Fill the byte array with the content of this message.

**See Also**        [Tibems.CreateFromBytes](#) on page 201

# Tibems.GetConnectAttempts

---

## Method

Declaration	<code>static string GetConnectAttempts();</code>
Purpose	Return the connection attempts setting.
Remarks	<p>This setting governs all client <code>Connection</code> objects as they attempt to connect to the server. Its value is a string of the form <i>attempts</i> or <i>attempts,delay</i>:</p> <ul style="list-style-type: none"><li><i>attempts</i> limits the number of times that the connection object attempts to establish a connection to the server. When this property is absent, the default value is 2. The minimum value is 1.</li><li><i>delay</i> is the time (in milliseconds) between connection attempts. When absent, the default value is 500. The minimum value is 250.</li></ul> <p>This method returns the string argument to <code>Tibems.SetConnectAttempts</code>—not the numeric value of the setting. If the client has not set a value, this method returns the null string.</p>
See Also	<code>Tibems.SetConnectAttempts</code> on page 218

## Tibems.GetEncoding

---

*Method*

<b>Declaration</b>	<code>static string GetEncoding();</code>
<b>Purpose</b>	Return the global character encoding for messages.
<b>Remarks</b>	<p>If the global encoding has not been set, this method returns null.</p> <p>Programs can override the global encoding for individual messages. When neither a global nor an individual message encoding has been set, then EMS encodes the strings of an outbound message using the default UTF-8 encoding.</p> <p>This encoding applies to all strings in message bodies (names and values), and properties (names and values). It does <i>not</i> apply to header names nor values. The methods <code>BytesMessage.ReadUTF</code> and <code>BytesMessage.WriteUTF</code> are exempt from global and individual encoding settings.</p>
<b>See Also</b>	<p><a href="#">BytesMessage—Read</a> on page 37</p> <p><a href="#">BytesMessage—Write</a> on page 40</p> <p><a href="#">Tibems.SetEncoding</a> on page 219</p> <p><a href="#">Tibems.GetMessageEncoding</a> on page 207</p> <p><a href="#">Tibems.SetMessageEncoding</a> on page 221</p>

## Tibems.GetExceptionOnFTSwitch

---

### Method

<b>Declaration</b>	<code>static bool GetExceptionOnFTSwitch();</code>
<b>Purpose</b>	Return the fault tolerance exception setting.
<b>Remarks</b>	<p>This setting determines exception behavior when the client successfully switches to a different server (fault-tolerant failover).</p> <ul style="list-style-type: none"><li>• When <code>true</code>, the connection's <code>ExceptionListener</code> catches an <code>EMSEException</code>, which contains the name of the new server.</li><li>• When <code>false</code>, fault-tolerant failover does not trigger an exception in the client.</li></ul>
<b>See Also</b>	<p><a href="#">IExceptionListener</a> on page 129</p> <p><a href="#">Tibems.SetExceptionOnFTSwitch</a> on page 220</p>

# Tibems.GetMessageEncoding

Method

Declaration	<code>static string GetMessageEncoding(     Message message );</code>
Purpose	Return the character encoding for an individual message.
Remarks	<p>If the encoding has not been set for the individual message, this method returns null.</p> <p>This encoding for an individual message overrides the global encoding. When neither a global nor an individual encoding has been set, then EMS encodes the strings of an outbound message using the default UTF-8 encoding.</p> <p>This encoding applies to all strings in message bodies (names and values), and properties (names and values). It does <i>not</i> apply to header names nor values. The methods <code>BytesMessage.ReadUTF</code> and <code>BytesMessage.WriteUTF</code> are exempt from global and individual encoding settings.</p>

Parameter	Description
message	Get the encoding for this message.

See Also	<div>BytesMessage—Read on page 37</div> <div>BytesMessage—Write on page 40</div> <div>Tibems.GetEncoding on page 205</div> <div>Tibems.SetEncoding on page 219</div> <div>Tibems.SetMessageEncoding on page 221</div>
----------	---

# Tibems.GetMessageSize

Method

Declaration

```
static int GetMessageSize(  
    Message msg );  
  
static int GetMessageBodySize(  
    Message msg );  
  
static int GetMessageHeadersSize(  
    Message msg );  
  
static int GetMessagePropertiesSize(  
    Message msg );
```

Purpose

Return the size of a wire format message—or its body, header or properties portions.

Remarks

These four methods return cached values for the size of a message or its parts.

The sizes are implicitly measured and cached when an inbound message arrives at the client, and when the client sends an outbound message. If the client modifies a message, or creates a message but never sends it, then these methods could yield incorrect cached values. To explicitly force a new measurement and cache its results, call `Tibems.CalculateMessageSize`; then these methods yield correct values.

`GetMessageSize` returns the total size of a message—that is, the number of bytes that traverse the network when the client sends the message. This total is slightly larger than the sum of its three constituent parts, because it includes additional control information. Furthermore, the server adds its own control information as well, so the size of message as measured by receivers is slightly larger than its size as measured by the sender.

Parameter	Description
msg	Return the cached size of this message, or one of its parts.

See Also

`Tibems.CalculateMessageSize` on page 200

## Tibems.GetPingInterval

---

*Method*

<b>Declaration</b>	<code>static int GetPingInterval();</code>
<b>Purpose</b>	Return the interval at which the client tests network connectivity.
<b>Remarks</b>	Clients test network connectivity by sending ping requests to the server at regular intervals. This method returns that interval (in seconds). Zero is a special value that disables ping testing.
<b>See Also</b>	<code>Tibems.SetPingInterval</code> on page 222

# Tibems.GetProperty

Method

- Declaration

static object GetProperty(  
    string key );
- Purpose

Return a property value.
- Remarks

If the property is not set, this method returns null.

In .NET, methods exist to get and set the properties. `GetProperty` provides an alternate way to get property values, which is consistent with the EMS Java API (for easy porting to .NET).

Parameter	Description
key	<p>Return the value associated with this property name.</p> <p>You may supply any of the constants listed in the table below. The constants are defined as static fields of <code>Tibems</code>. The values of those constants are the actual property names.</p>

(Sheet 1 of 2)

Property Constant	Corresponding Methods
PROP_SOCKET_RECEIVE_SIZE	<p><code>Tibems.GetSocketReceiveBufferSize</code> on page 214</p> <p><code>Tibems.SetSocketReceiveBufferSize</code> on page 227</p>
PROP_SOCKET_SEND_SIZE	<p><code>Tibems.GetSocketSendBufferSize</code> on page 215</p> <p><code>Tibems.SetSocketSendBufferSize</code> on page 228</p>
PROP_CONNECTION_ATTEMPTS	<p><code>Tibems.GetConnectAttempts</code> on page 204</p> <p><code>Tibems.SetConnectAttempts</code></p>
PROP_RECONNECTION_ATTEMPTS	<p><code>Tibems.GetReconnectAttempts</code> on page 212</p> <p><code>Tibems.SetReconnectAttempts</code> on page 225</p>
PROP_CLOSE_IN_CALLBACK	<p><code>Tibems.GetAllowCloseInCallback</code> on page 202</p> <p><code>Tibems.SetAllowCloseInCallback</code> on page 217</p>



(Sheet 2 of 2)

Property Constant	Corresponding Methods
PROP_PING_INTERVAL	Tibems.GetPingInterval on page 209 Tibems.SetPingInterval on page 222
PROP_FT_SWITCH	Tibems.GetExceptionOnFTSwitch on page 206 Tibems.SetExceptionOnFTSwitch on page 220
PROP_MESSAGE_ENCODING	Tibems.GetMessageEncoding on page 207 Tibems.SetMessageEncoding on page 221
PROP_DAEMON_DISPATCHER	Tibems.GetSessionDispatcherDaemon on page 213 Tibems.SetSessionDispatcherDaemon on page 226

**See Also**    Tibems.SetProperty on page 223

# Tibems.GetReconnectAttempts

Method

Declaration	<code>static string GetReconnectAttempts();</code>
Purpose	Return the reconnection attempts setting.
Remarks	<p>This setting governs all client <code>Connection</code> objects as they attempt to reconnect to the server after a network disconnect. Its value is a string of the form <i>attempts</i> or <i>attempts, delay</i>:</p> <ul style="list-style-type: none"><li><i>attempts</i> limits the number of times that the connection object attempts to reestablish a connection to the server. When this property is absent, the default value is 4. The minimum value is 1.</li><li><i>delay</i> is the time (in milliseconds) between reconnection attempts. When absent, the default value is 500. The minimum value is 250.</li></ul> <p>This method returns the string argument to <code>Tibems.SetReconnectAttempts</code>—not the numeric value of the setting. If the client has not set a value, this method returns the null string.</p>
See Also	<code>Tibems.SetReconnectAttempts</code> on page 225

## Tibems.GetSessionDispatcherDaemon

---

*Method*

<b>Declaration</b>	<code>static bool GetSessionDispatcherDaemon();</code>
<b>Purpose</b>	Return the dispatcher thread setting.
<b>Remarks</b>	When a program uses asynchronous message consumers (either message listeners or .NET message event handler delegates), EMS creates internal dispatcher threads for each <code>Session</code> that has at least one asynchronous message consumer. When this setting is <code>true</code> , those dispatcher threads are daemon threads; when <code>false</code> (the default) they are not daemon threads.
<b>.NET Compact Framework</b>	The .NET Compact Framework does not support daemon threads. This call always returns <code>false</code> .
<b>See Also</b>	<code>Tibems.SetSessionDispatcherDaemon</code> on page 226

# Tibems.GetSocketReceiveBufferSize

---

Method

Declaration	<code>static int GetSocketReceiveBufferSize();</code>
Purpose	Return the size of socket receive buffers.
Remarks	When set, this value overrides the operating system’s default for the size of receive buffers associated with sockets that the client uses for connections to the server. (Some operating systems do not allow you to override the default size.)
See Also	<code>Tibems.SetSocketReceiveBufferSize</code> on page 227

## Tibems.GetSocketSendBufferSize

---

*Method*

<b>Declaration</b>	<code>static int GetSocketSendBufferSize();</code>
<b>Purpose</b>	Return the size of socket send buffers.
<b>Remarks</b>	When set, this value overrides the operating system's default for the size of send buffers associated with sockets that the client uses for connections to the server. (Some operating systems do not allow you to override the default size.)
<b>See Also</b>	<code>Tibems.SetSocketSendBufferSize</code> on page 228

# Tibems.MakeWriteable

Method

**Declaration**     `static void MakeWriteable(  
                         Message message);`

**Purpose**            Make a message writeable.

Parameter	Description
message	Make this message writeable.

**See Also**        [MessageNotWriteableException](#) on page 244

# Tibems.SetAllowCloseInCallback

## Method

<b>Declaration</b>	<code>static void SetAllowCloseInCallback( bool allow );</code>
<b>Purpose</b>	Override a JMS requirement so client callbacks may call close methods.
<b>Remarks</b>	<p>According to the JMS specification, close methods (that is, <code>MessageConsumer.Close</code>, <code>Session.Close</code>, <code>Connection.Close</code>) cannot return while any message callbacks (that is, <code>EMSMessagesHandler.OnMessage</code>, <code>IMessageListener.OnMessage</code>) are running. As a result, a message callback must not call a close method, lest it cause a deadlock.</p> <p>This method explicitly overrides this JMS requirement, permitting callbacks to call close without deadlock (that is, embedded close calls do not wait for callbacks to return).</p>

Parameter	Description
allow	<p>When <code>true</code>, EMS overrides the JMS specification.</p> <p>When <code>false</code> (the default), EMS obeys the JMS specification.</p>



This method replaces the deprecated method `SetAllowCallbackInClose`.

**See Also** `Tibems.GetAllowCloseInCallback` on page 202

# Tibems.SetConnectAttempts

## Method

Declaration	static void SetConnectAttempts( string specs );
Purpose	Modify the connection attempts setting.
Remarks	This setting governs all client Connection objects as they attempt to connect to the server.

Parameter	Description
specs	Set the connect setting to these specifications. The value must be string of the form <i>attempts</i> or <i>attempts , delay</i> : <ul style="list-style-type: none"><li><i>attempts</i> limits the number of times that the connection object attempts to establish a connection to the server. When this property is absent, the default value is 2. The minimum value is 1.</li><li><i>delay</i> is the time (in milliseconds) between connection attempts. When absent, the default value is 500. The minimum value is 250.</li></ul>

**See Also**    Tibems.GetConnectAttempts on page 204



## Tibems.SetEncoding

Method

**Declaration**     `static void SetEncoding(  
                         string encodingName );`

**Purpose**            Set the global character encoding for messages.

**Remarks**        Programs can override the global encoding for individual messages. When neither a global nor an individual message encoding has been set, then EMS encodes the strings of an outbound message using the default UTF-8 encoding.

This encoding applies to all strings in message bodies (names and values), and properties (names and values). It does *not* apply to header names nor values. The methods `BytesMessage.ReadUTF` and `BytesMessage.WriteUTF` are exempt from global and individual encoding settings.

Parameter	Description
encodingName	Set this global encoding.  For a list of standard encoding names, see <a href="http://www.iana.org">www.iana.org</a> .

**See Also**        `BytesMessage`—Read on page 37  
                  `BytesMessage`—Write on page 40  
                  `Tibems.GetEncoding` on page 205  
                  `Tibems.GetMessageEncoding` on page 207  
                  `Tibems.SetMessageEncoding` on page 221

# Tibems.SetExceptionOnFTSwitch

## Method

- Declaration

static void SetExceptionOnFTSwitch(  
    bool callExceptionListener );
- Purpose

Modify the fault tolerance exception setting.
- Remarks

This setting determines exception behavior when the client successfully switches to a different server (fault-tolerant failover).

Parameter	Description
callExceptionListener	When true, the connection’s ExceptionListener catches an EMSEException, which contains the name of the new server.  When false, fault-tolerant failover does not trigger an exception in the client.

- See Also

IExceptionListener on page 129

Tibems.GetExceptionOnFTSwitch on page 206

## Tibems.SetMessageEncoding

### Method

**Declaration**     `static void SetMessageEncoding(  
                    Message message  
                    string encodingName );`

**Purpose**            Set the character encoding for an individual message.

**Remarks**        This encoding for an individual message overrides the global encoding. When neither a global nor an individual message encoding has been set, then EMS encodes the strings of an outbound message using the default UTF-8 encoding.

This encoding applies to all strings in message bodies (names and values), and properties (names and values). It does *not* apply to header names nor values. The methods `BytesMessage.ReadUTF` and `BytesMessage.WriteUTF` are exempt from global and individual encoding settings.

Parameter	Description
message	Set the encoding for this message.
encodingName	Set this encoding.  For a list of standard encoding names, see <a href="http://www.iana.org">www.iana.org</a> .

**See Also**        `BytesMessage`—Read on page 37  
                  `BytesMessage`—Write on page 40  
                  `Tibems.GetEncoding` on page 205  
                  `Tibems.SetEncoding` on page 219  
                  `Tibems.GetMessageEncoding` on page 207

# Tibems.SetPingInterval

Method

- Declaration

static void SetPingInterval(  
int seconds );
- Purpose

Set the interval at which the client tests network connectivity.
- Remarks

Clients test network connectivity by sending ping requests to the server at regular intervals. This method sets that interval (in seconds). If your program calls this method, it must do so before creating its first `Connection` object; after creating that object, this call has no effect.

Parameter	Description
seconds	Ping at this interval (in seconds).  Zero is a special value that disables ping testing.

See Also

Tibems.GetPingInterval on page 209

## Tibems.SetProperty

### Method

**Declaration**     `static void SetProperty(  
                    string key,  
                    object val );`

**Purpose**            Modify a property value.

**Remarks**        In .NET, methods exist to get and set the properties. SetProperty provides an alternate way to set property values, which is consistent with the EMS Java API (for easy porting to .NET).

Parameter	Description
key	Set the value associated with this property name.  You may supply any of the constants listed in the table below. The constants are defined as static fields of Tibems. The values of those constants are the actual property names.
value	Set the property to this value.

(Sheet 1 of 2)

Property Constant	Corresponding Methods
PROP_SOCKET_RECEIVE_SIZE	Tibems.GetSocketReceiveBufferSize on page 214 Tibems.SetSocketReceiveBufferSize on page 227
PROP_SOCKET_SEND_SIZE	Tibems.GetSocketSendBufferSize on page 215 Tibems.SetSocketSendBufferSize on page 228
PROP_CONNECTION_ATTEMPTS	Tibems.GetConnectAttempts on page 204 Tibems.SetConnectAttempts
PROP_RECONNECTION_ATTEMPTS	Tibems.GetReconnectAttempts on page 212 Tibems.SetReconnectAttempts on page 225
PROP_CLOSE_IN_CALLBACK	Tibems.GetAllowCloseInCallback on page 202 Tibems.SetAllowCloseInCallback on page 217

(Sheet 2 of 2)

Property Constant	Corresponding Methods
PROP_PING_INTERVAL	Tibems.GetPingInterval on page 209 Tibems.SetPingInterval on page 222
PROP_FT_SWITCH	Tibems.GetExceptionOnFTSwitch on page 206 Tibems.SetExceptionOnFTSwitch on page 220
PROP_MESSAGE_ENCODING	Tibems.GetMessageEncoding on page 207 Tibems.SetMessageEncoding on page 221
PROP_DAEMON_DISPATCHER	Tibems.GetSessionDispatcherDaemon on page 213 Tibems.SetSessionDispatcherDaemon on page 226

**See Also**    Tibems.GetProperty on page 210

## Tibems.SetReconnectAttempts

### Method

<b>Declaration</b>	<code>static void SetReconnectAttempts(     string specs );</code>
<b>Purpose</b>	Modify the reconnection attempts setting.
<b>Remarks</b>	This setting governs all client <code>Connection</code> objects as they attempt to reconnect to the server after a network disconnect.

Parameter	Description
specs	<p>Set the reconnect setting to these specifications. The value must be string of the form <i>attempts</i> or <i>attempts , delay</i>:</p> <ul style="list-style-type: none"><li>• <i>attempts</i> limits the number of times that the connection object attempts to reestablish a connection to the server. When this property is absent, the default value is 4. The minimum value is 1.</li><li>• <i>delay</i> is the time (in milliseconds) between reconnection attempts. When absent, the default value is 500. The minimum value is 250.</li></ul>

**See Also**    `Tibems.GetReconnectAttempts` on page 212

# Tibems.SetSessionDispatcherDaemon

Method

- Declaration

static void SetSessionDispatcherDaemon(  
bool makeDaemon );
- Purpose

Set the dispatcher thread setting.
- Remarks

When a program uses asynchronous message consumers (either message listeners or .NET message event handler delegates), EMS creates internal dispatcher threads for each Session that has at least one asynchronous message consumer. When this setting is true, those dispatcher threads are daemon threads; when false (the default) they are not daemon threads.

Parameter	Description
makeDaemon	<ul style="list-style-type: none"><li>When true, dispatcher threads are daemon threads.</li><li>When false, (the default) they are not daemon threads</li></ul>

- .NET Compact Framework

The .NET Compact Framework does not support daemon threads. This call has no effect, and returns without error.
- See Also

Tibems.GetSessionDispatcherDaemon on page 213



# Tibems.SetSocketReceiveBufferSize

Method

<b>Declaration</b>	<code>static void SetSocketReceiveBufferSize( int size );</code>
<b>Purpose</b>	Set the size of socket receive buffers.
<b>Remarks</b>	<p>This value overrides the operating system’s default for the size of receive buffers associated with sockets that the client uses for connections to the server.</p> <p>Use this call before creating server connections. This call sets an override buffer size for new socket buffers; it does not change the size of existing socket buffers.</p>

Parameter	Description
size	Sockets use receive buffers of this size (in kilobytes).

<b>.NET Compact Framework</b>	The .NET Compact Framework does not permit changing the default socket buffer size. This call has no effect, and returns without error.
<b>See Also</b>	<code>Tibems.GetSocketReceiveBufferSize</code> on page 214

# Tibems.SetSocketSendBufferSize

Method

- Declaration

static void SetSocketSendBufferSize(  
int size );
- Purpose

Set the size of socket send buffers.
- Remarks

This value overrides the operating system’s default for the size of send buffers associated with sockets that the client uses for connections to the server.

Use this call before creating server connections. This call sets an override buffer size for new socket buffers; it does not change the size of existing socket buffers.

Parameter	Description
size	Sockets use send buffers of this size (in kilobytes).

- .NET Compact Framework

The .NET Compact Framework does not permit changing the default socket buffer size. This call has no effect, and returns without error.
- See Also

Tibems.GetSocketSendBufferSize on page 215

## Chapter 14    **Exception**

This chapter presents exceptions related to EMS.

### Topics

---

- *EMSEException, page 230*
- *AuthenticationException, page 232*
- *CannotProceedException, page 233*
- *CommunicationException, page 234*
- *ConfigurationException, page 235*
- *IllegalStateException, page 236*
- *InvalidClientIDException, page 237*
- *InvalidDestinationException, page 238*
- *InvalidNameException, page 239*
- *InvalidSelectorException, page 240*
- *MessageEOFException, page 241*
- *MessageFormatException, page 242*
- *MessageNotReadableException, page 243*
- *MessageNotWriteableException, page 244*
- *NameNotFoundException, page 245*
- *NamingException, page 246*
- *ResourceAllocationException, page 247*
- *SecurityException, page 248*
- *ServiceUnavailableException, page 249*
- *TransactionInProgressException, page 250*
- *TransactionRolledBackException, page 251*

# EMSException

Class

- Declaration

class EMSException : System.Exception
- Purpose

Root of exceptions specific to EMS.
- Origin

Corresponds to JMSException in JMS.
- Remarks

EMS methods throw instances of this class and its subclasses.

Member	Description
Properties	
ErrorCode	<div>string {get;}</div> <div>When an exception results from a server error, this property holds the server's error code.</div>
LinkedException	<div>System.Exception {get; set;}</div> <div>When an EMS exception results from a deeper problem, this linked exception details that problem.</div>

- Subclasses

EMSException

IllegalStateException

InvalidClientIDException

InvalidDestinationException

InvalidSelectorException

MessageEOFException

MessageFormatException

MessageNotReadableException

MessageNotWriteableException

NamingException

AuthenticationException

CannotProceedException

CommunicationException

ConfigurationException

InvalidNameException

NameNotFoundException

ServiceUnavailableException

ResourceAllocationException

SecurityException

TransactionInProgressException

TransactionRolledBackException

**Constructors**

```
EMSEException (
    string reason );

EMSEException (
    string reason,
    string errorCode );
```

# AuthenticationException

---

*Class*

- Declaration**    `class AuthenticationException : NamingException`
- Purpose**        Invalid authentication or insufficient privileges for a lookup request.
- See Also**      `LookupContext` on page 182

# CannotProceedException

---

*Class*

<b>Declaration</b>	<code>class CannotProceedException : NamingException</code>
<b>Purpose</b>	Insufficient information to resolve a lookup request.
<b>Remarks</b>	<p>A destination lookup request found both a queue and a topic with the specified name. To resolve this situation, specify the destination name in one of these forms:</p> <ul style="list-style-type: none"><li>• <code>\$topic:&lt;topic-name&gt;</code></li><li>• <code>\$queue:&lt;queue-name&gt;</code></li></ul>
<b>See Also</b>	LookupContext on page 182

# CommunicationException

---

Class

Declaration	class CommunicationException : NamingException
Purpose	A lookup request returned bad data.
Remarks	This exception could indicate a version mismatch between the client and tibemsd.
See Also	LookupContext on page 182



# ConfigurationException

---

*Class*

<b>Declaration</b>	<code>class ConfigurationException : NamingException</code>
<b>Purpose</b>	Configuration error associated with a lookup context object.
<b>Remarks</b>	When the client initialized the lookup context, some parameter values were missing or invalid.
<b>See Also</b>	LookupContext on page 182

# IllegalStateException

---

Class

Declaration	<code>class IllegalStateException : EMSEException</code>
Purpose	A method call or server request occurred in an inappropriate context.
Origin	JMS.
Remarks	For example, <code>Session.Commit</code> throws this exception when the session is non-transactional.

## InvalidClientIDException

---

*Class*

<b>Declaration</b>	<code>class InvalidClientIDException : EMSEException</code>
<b>Purpose</b>	The provider rejects the connection's client ID.
<b>Origin</b>	JMS.
<b>Remarks</b>	Setting a connection's client ID to an invalid or duplicate value results in this exception. (A duplicate value is one that is already in use by another connection.)

# InvalidDestinationException

---

Class

Declaration	class InvalidDestinationException : EMSEException
Purpose	tibemsd cannot locate the destination.
Origin	JMS.

# InvalidNameException

---

*Class*

<b>Declaration</b>	<code>class InvalidNameException : NamingException</code>
<b>Purpose</b>	In a lookup request, the name has incorrect syntax.
<b>Remarks</b>	The most common syntax error is a prefix other than <code>tibjmsnaming://</code> (or a misspelling).
<b>See Also</b>	<code>LookupContext</code> on page 182

# InvalidSelectorException

---

*Class*

- Declaration**     `class InvalidSelectorException : EMSEException`
- Purpose**           The client passed a message selector with invalid syntax.
- Origin**            JMS.
- See Also**         Message Selectors on page 20

## MessageEOFException

---

*Class*

<b>Declaration</b>	<code>class MessageEOFException : EMSEException</code>
<b>Purpose</b>	The data stream within a message ended unexpectedly.
<b>Origin</b>	JMS.
<b>Remarks</b>	<code>BytesMessage</code> contains a stream of bytes. <code>StreamMessage</code> contains a stream of characters. If any of their read methods detects the end of the stream unexpectedly, it throws this exception.

# MessageFormatException

---

Class

Declaration	<code>class MessageFormatException : EMSEException</code>
Purpose	Datatype mismatch.
Origin	JMS.
Remarks	<div>For example:<ul style="list-style-type: none"><li>• A read method cannot read the data with the specified type.</li><li>• A write method cannot store the data in the message or property because the data has the wrong type.</li></ul></div>



## MessageNotReadableException

---

*Class*

**Declaration**    `class MessageNotReadableException : EMSEException`

**Purpose**        Attempt to read from a message in write-only mode.

**Origin**        JMS.

# MessageNotWriteableException

---

*Class*

<b>Declaration</b>	<code>class MessageNotWriteableException : EMSEException</code>
<b>Purpose</b>	Attempt to write to a message in read-only mode.
<b>Origin</b>	JMS.
<b>See Also</b>	<code>Tibems.MakeWriteable</code> on page 216

# NameNotFoundException

---

*Class*

**Declaration**    `class NameNotFoundException : NamingException`

**Purpose**        The name lookup repository cannot find a name; the name is not bound.

**See Also**      [LookupContext](#) on page 182

# NamingException

Class

- Declaration**`class NamingException : EMSEException`
- Purpose**Root of exceptions related to name lookup requests.

Member	Description
Properties	
RootCause	<code>System.Exception {get; set;}</code> When a naming exception results from a more general problem, this exception details that problem.

- Remarks**Members of `LookupContext` throw instances of this class and its subclasses.
- Subclasses**`NamingException`  
`AuthenticationException`  
`CannotProceedException`  
`CommunicationException`  
`ConfigurationException`  
`InvalidNameException`  
`NameNotFoundException`  
`ServiceUnavailableException`
- See Also**`LookupContext` on page 182

## ResourceAllocationException

---

*Class*

**Declaration**    `class ResourceAllocationException : EMSEException`

**Purpose**        Required resources are not available.

**Origin**        JMS.

# SecurityException

---

Class

Declaration	class SecurityException : EMSEException
Purpose	The method cannot complete because of a security restriction.
Origin	JMS.
Remarks	For example, the provider rejects a user or the user’s authentication.

## ServiceUnavailableException

---

*Class*

**Declaration**    `class ServiceUnavailableException : NamingException`

**Purpose**        A lookup request failed because the client could not connect to the server.

**See Also**      [LookupContext](#) on page 182

# TransactionInProgressException

Class

Declaration	<code>class TransactionInProgressException : EMSEException</code>
Purpose	Reserved for future use.
Origin	JMS.
Remarks	When a session uses an XA transaction manager, the XA resource is the correct locus for all commit and rollback requests. Local commit or rollback calls are not permitted, and throw this exception.



## TransactionRolledBackException

---

*Class*

**Declaration**    `class TransactionRolledBackException : EMSException`

**Purpose**        An attempt to commit a transaction resulted in rollback.

**Origin**        JMS.



# Index

## A

Acknowledge 27  
 assembly 7  
 AuthenticationException 232

## B

body types, message 11  
 BodyLength 36  
 BytesMessage 36  
   Read methods 37  
   ReadBytes 39  
   Reset 43  
   Write methods 40  
   WriteBytes 42

## C

cache, assembly 7  
 CalculateMessageSize 200  
 CannotProceedException 233  
 character encoding 4, 205, 207, 219, 221  
 checklist, programmer's 7  
 ClearBody 28  
 ClearProperties 29  
 Clone 30  
 Close  
   Connection 117  
   MessageConsumer 78  
   MessageProducer 93  
   QueueBrowser 176  
   QueueRequestor 106  
   Session 148  
   TopicRequestor 110

Commit 149  
 CommunicationException 234  
 Compact Framework 5  
 compression 3  
 ConfigurationException 235  
 Connection 114  
   Close 117  
   CreateSession 118  
   Start 119  
   Stop 120  
 ConnectionConsumer (not supported) 3  
 ConnectionFactory 132  
   constructor 134  
   CreateConnection 135  
 ConnectionMetaData 121  
 conversion, data type 23  
 CreateBrowser 150  
 CreateBytesMessage 151  
 CreateConnection 135  
 CreateConsumer 152  
 CreateDurableSubscriber 153  
 CreateFromBytes 201  
 CreateMapMessage 155  
 CreateObjectMessage 156  
 CreateProducer 157  
 CreateQueue 158  
 CreateQueueConnection 138  
 CreateQueueSession 123  
 CreateSession 118  
 CreateStreamMessage 159  
 CreateTemporaryQueue 160  
 CreateTemporaryTopic 161  
 CreateTextMessage 162  
 CreateTopic 163  
 CreateTopicConnection 140  
 CreateTopicSession 125  
 customer support xiv

**D**

- daemon threads 3
- data type conversion 23
- Delete
  - TemporaryQueue 69
  - TemporaryTopic 71
- DeliveryMode 190
- Deserialize 192
- Destination 65
  - overview 62
- durable subscription
  - unsubscribe 167
- dynamic destination 62

**E**

- EMSException 230
- EMSExceptionEventArgs 127
  - constructor 128
- EMSExceptionHandler 126
- EMSMessageEventArgs 84
  - constructor 85
  - Message 84
- EMSMessageHandler 83
- encoding, character 4, 205, 207, 219, 221
- exceptions 229

**F**

- FactoryLoadBalanceMetric 136
- fault tolerance
  - ActiveURL 115
  - failover exceptions 206, 220
- FieldCount
  - MapMessage 44
  - StreamMessage 52

**G**

- Get
  - MapMessage 46
  - message properties 32
- GetAllowCloseInCallback 202
- GetAsBytes 203
- GetConnectAttempts 204
- GetDeliveryModeName 31
- GetEncoding 205
- GetEnumerator 177
- GetExceptionOnFTSwitch 206
- GetMessageBodySize 208
- GetMessageEncoding 207
- GetMessageHeadersSize 208
- GetMessageSize 208
- GetPingInterval 209
- GetProperty, Tibems 210
- GetReconnectAttempts 212
- GetSessionDispatcherDaemon 213
- GetSocketReceiveBufferSize 214
- GetSocketSendBufferSize 215
- global assembly cache 7

**H**

- headers, message 12

**I**

- IEMSSerializable 191
  - Deserialize 192
  - Serialize 193
- IExceptionHandler 129
  - OnException 130
- IllegalStateException 236
- IMessageListener 86
  - OnMessage 87
- InvalidClientIDException 237
- InvalidDestinationException 238
- InvalidNameException 239

InvalidSelectorException 240  
ItemExists 47

## L

Lookup 186  
LookupContext 182  
    AddSettings  
        AddSettings 185  
    constructor 184  
    Lookup 186  
    RemoveSettings 187

## M

MakeWriteable 216  
MapMessage 44  
    FieldCount 44  
    Get methods 46  
    ItemExists 47  
    MapNames 44  
    set methods 48  
    SetBytes 49  
MapNames 44  
Message 24  
    Acknowledge 27  
    body types 11  
    BodyLength 36  
    ClearBody 28  
    ClearProperties 29  
    Clone 30  
    EMSMessageEventArgs 84  
    get property methods 32  
    GetDeliveryModeName 31  
    headers 12  
    parts of 10  
    properties 17  
    PropertyExists 33  
    set property methods 34  
    ToString 35

message  
    selectors 20  
MessageConsumer 76  
    Close 78  
    MessageHandler 76  
    MessageListener 76  
    MessageSelector 77  
    Receive 79  
    ReceiveNoWait 80  
MessageDeliveryMode 194  
MessageEOFException 241  
MessageFormatException 242  
MessageHandler 76  
MessageListener 76  
MessageNotReadableException 243  
MessageNotWriteableException 244  
MessageProducer 90  
    Close 93  
    Send 94  
MessageProducer.Close 93  
MessageProducer.Send 94  
MessageSelector 77  
MoveNext 178

## N

NameNotFoundException 245  
NamingException 246  
.NET Compact Framework 5  
NoLocal 82

## O

object serialization 3  
ObjectMessage 50  
    constructor 51  
    TheObject 50  
OnException 130  
OnMessage 87

**P**

property, message 17  
     get 32  
     set 34  
 PropertyExists 33  
 Publish 100

**Q**

Queue 66  
     constructor 67  
     QueueName 66  
     QueueReceiver 81  
 QueueBrowser 174  
     Close 176  
     GetEnumerator 177  
     MoveNext 178  
     Reset 179  
 QueueConnection 122  
     CreateQueueSession 123  
 QueueConnectionFactory 137  
     CreateQueueConnection 138  
 QueueName 66  
 QueueReceiver 81  
     Queue 81  
 QueueRequestor 104  
     Close 106  
     constructor 105  
     Request 107  
 QueueSender 96  
     Send 97  
 QueueSender.Send 97  
 QueueSession 170

**R**

Read  
     BytesMessage 37  
     StreamMethod 54

ReadBytes  
     BytesMessage 39  
     StreamMessage 55  
 read-only 28, 43, 56, 201, 244  
 Receive 79  
 ReceiveNoWait 80  
 Recover 164  
 RemoveSettings 187  
 Request  
     QueueRequestor 107  
     TopicRequestor 111  
 request 12  
 Reset  
     BytesMessage 43  
     QueueBrowser 179  
     StreamMessage 56  
 ResourceAllocationException 247  
 Rollback 165  
 Run (obsolete) 166

**S**

SecurityException 248  
 selectors, message 20  
 Send  
     MessageProducer 94  
     QueueSender 97  
 serializable object interface 191  
 serialization 3  
 Serialize 193  
 ServerSession (not supported) 3  
 ServerSessionPool (not supported) 3  
 ServiceUnavailableException 249

- Session 142
  - Close 148
  - Commit 149
  - CreateBrowser 150
  - CreateBytesMessage 151
  - CreateConsumer 152
  - CreateDurableSubscriber 153
  - CreateMapMessage 155
  - CreateObjectMessage 156
  - CreateProducer 157
  - CreateQueue 158
  - CreateStreamMessage 159
  - CreateTemporaryQueue 160
  - CreateTemporaryTopic 161
  - CreateTextMessage 162
  - CreateTopic 163
  - Recover 164
  - Rollback 165
  - Run (obsolete) 166
  - Unsubscribe 167
- SessionMode 168
- Set
  - MapMessage 48
  - message property 34
- SetAllowCloseInCallback 217
- SetBytes 49
- SetConnectAttempts 218
- SetEncoding 219
- SetExceptionOnFTSwitch 220
- SetMessageEncoding 221
- SetPingInterval 222
- SetProperty, Tibems 223
- SetReconnectAttempts 225
- SetSessionDispatcherDaemon 226
- SetSocketReceiveBufferSize 227
- SetSocketSendBufferSize 228
- SSL 3
- Start 119
- static destination 62
- Stop 120

- StreamMessage 52
  - FieldCount 52
  - Read methods 54
  - ReadBytes 55
  - Reset 56
  - Write methods 57
  - WriteBytes 58
- string and character encoding 4
- support, contacting xiv

## T

- technical support xiv
- temporary destination 62
- TemporaryQueue 68
  - Delete 69
- TemporaryTopic 70
  - Delete 71
- Text 59
- TextMessage 59
  - constructor 60
  - Text 59
- TheObject 50

## Tibems 195

- CalculateMessageSize 200
- CreateFromBytes 201
- GetAllowCloseInCallback 202
- GetAsBytes 203
- GetConnectAttempts 204
- GetEncoding 205
- GetExceptionOnFTSwitch 206
- GetMessageBodySize 208
- GetMessageEncoding 207
- GetMessageHeadersSize 208
- GetMessageSize 208
- GetPingInterval 209
- GetProperty 210
- GetReconnectAttempts 212
- GetSessionDispatcherDaemon 213
- GetSocketReceiveBufferSize 214
- GetSocketSendBufferSize 215
- MakeWriteable 216
- SetAllowCloseInCallback 217
- SetConnectAttempts 218
- SetEncoding 219
- SetExceptionOnFTSwitch 220
- SetMessageEncoding 221
- SetPingInterval 222
- SetProperty 223
- SetReconnectAttempts 225
- SetSessionDispatcherDaemon 226
- SetSocketReceiveBufferSize 227
- SetSocketSendBufferSize 228

tibemsd 7, 132, 181

## Topic 72

- constructor 73
- TopicName 72
- TopicSubscriber 82

## TopicConnection 124

- CreateTopicSession 125

## TopicConnectionFactory 139

- CreateTopicConnection 140

## TopicName 72

## TopicPublisher 99

- Publish 100

## TopicPublisher.Publish 100

## TopicRequestor 108

- Close 110
- constructor 109
- Request 111

## TopicSession 171

## TopicSubscriber 82

- NoLocal 82
- Topic 82

## ToString 35

## TransactionInProgressException 250

## TransactionRolledBackException 251

## translation, character encoding 4

## type conversion 23

## U

## Unicode 4

## Unsubscribe 167

## W

## Write

- BytesMessage 40
- StreamMessage 57

## WriteBytes

- BytesMessage 42
- StreamMessage 58

## X

## XA (not supported) 3