Cincom®

Cincom Smalltalk™

VisualWorks®

MQ-Interface Guide

VisualWorks 8.0

P46-0148-02

SIMPLIFICATION THROUGH INNOVATION

Part Number: P46-0148-02

Software Release 8.0

This document is subject to change without notice.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About This Book

Overview

This document describes the usage of the VisualWorks interface to WebSphere® MQ using the bshared libraries provided by IBM. WebSphere MQ is a message base communication system by which two or more application can exchange messages through a queue. It is like electronic mail between applications.

The MQ-Interface provides an easy-to-use interface for VisualWorks to connect to the WebSphere MQ shared libraries. It simplifies the tedious handling of the C-call interface and its parameter. A developer can use WebSphere MQ through a simple and efficient feature wrapper. This document describes the feature wrapper, its classes and methods, and how to use it.

Audience

This guide assumes that you already know VisualWorks and Smalltalk, and are familiar with WebSphere MQ.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.

Example	Description
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > New	Indicates the name of an item (**New**) on a menu (**File**).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help > About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help > About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **Copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to:

supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Personal Use License (PUL) is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides an important public resource for VisualWorks developers:

- A mailing list for users of VisualWorks PUL, which serves a growing community of users. To subscribe or unsubscribe, send a message to: vwnc-request@cs.uiuc.edu.

  with the SUBJECT of **subscribe** or **unsubscribe**. You can then address emails to: vwnc@cs.uiuc.edu.

The Smalltalk news group, comp.lang.smalltalk, carries on general discussions about different dialects of Smalltalk, including VisualWorks, and is a good source for advice.

Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks:

> http://www.cincomsmalltalk.com/main/products/visualworks/visualworks-tutorials/

The guide you are reading is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

> http://www.cincomsmalltalk.com/main/products/documentation/

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Online Help

VisualWorks includes an online help system. To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks Launcher window, and select one of the help options.

1

WebSphere MQ

WebSphere MQ is an IBM product family that enables applications to exchange information across different platforms, sending and receiving messages through queues. The main feature is its loose coupling of applications; one application does not depend on the state of another application, it does not need to know if the other application is ready to receive a message.

An application just writes a message to a queue. The receiving application retrieves the message when it is ready, processes the request, and writes a reply message to a queue if necessary. The requesting application can actively wait for a replay. It also can setup a separate process that waits and proceeds with other tasks.

Architectures

The general architecture of WebSphere MQ connects an application via a channel to a queue manager. The application opens a queue, and then either writes messages to it or reads messages from it. Queues are always unidirectional; it is not possible to open a single queue for sending and receiving.

WebSphere MQ provides different setups for various application configurations. Here we describe the three most common setups. The main difference is the location of the queue manager. The manager can be installed on the same machine as the application(s) that will exchange information.

Basically all setup requires only a small set of functionality:

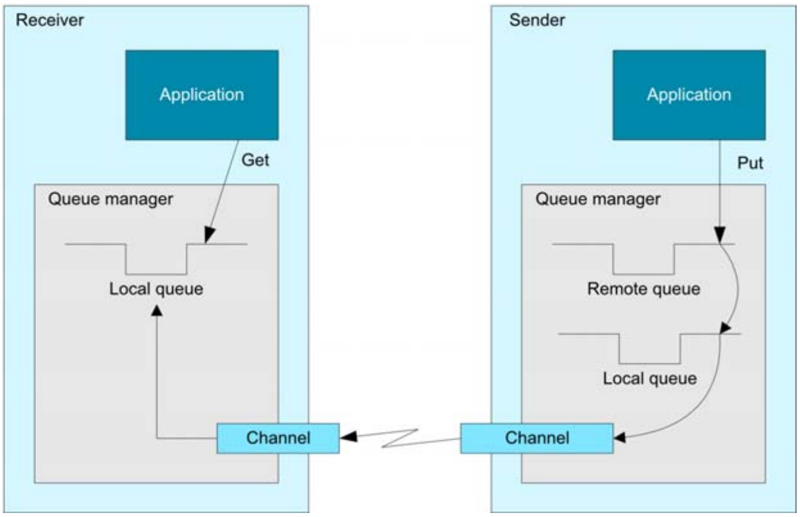
- Connect to a queue manager (local or remote)
- Open a queue (receiver or sender / publisher or subscriber)

- Put a message to the queue (sender or publisher)
- Get a message from the queue (receiver or subscriber)
- Close a queue
- Disconnect from the queue manager

The setup only defines how an application connects to a queue manager and how it opens a queue. Retrieving and writing messages does not depend on the setup. They depend on what type of messages an application want to send or retrieve.

Local Queue Setup

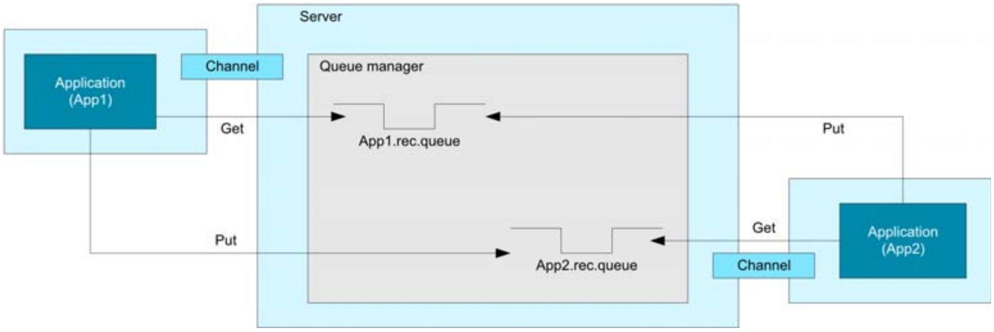
In a local setup, queue managers are connected by a channel, usually via TCP/IP.



The sender's remote queue knows the name of the receiver's queue manager and queue. The remote queue does not know where the receiver's queue is, but the sender's local queue knows its channel and the channel knows the receiver's queue manager. The queues are coupled together. Any message the sender application writes to its remote queue will be transferred to the local queue of the receiver.

Remote Queue Setup

The most common setup uses a dedicated server for the queue manager and all (or most) applications connect to this queue manager as clients.

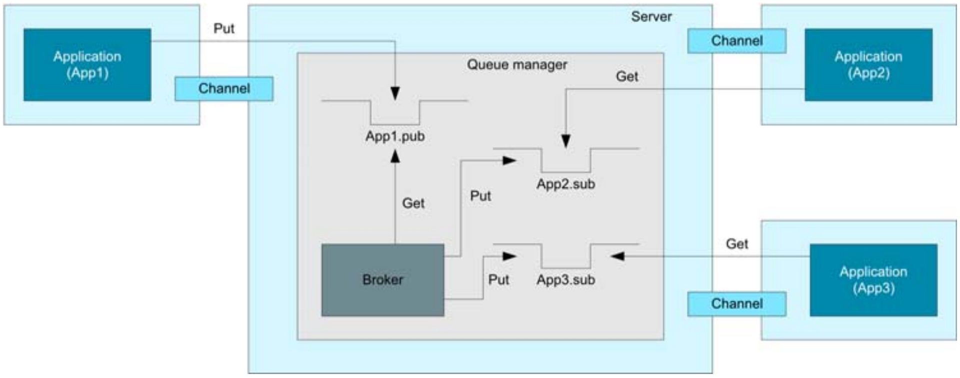


The clients open the receiver queue of that application they want to send a message and put the message into that queue. In this setup there is no dedicated sender queue. The clients only read from their receiver queues.

In the above setup an application must know the name of the receiver for the other application. Therefore, an application must control a queue for each application to which it wants to send messages.

Subscriber/Publisher Setup

Using the WebSphere MQ integrator you can setup a network topology where the central instance, the WebSphere MQ Broker, distributes the messages.



It can use defined parameters in the message, or it can filter the contents to determine where the message will be routed. If an application wants to send data it opens a publisher queue and put the message into this queue. If an application wants to retrieve messages it opens a subscriber queue and waits (GET) until data arrives.

Message Types

WebSphere MQ defines four types of messages:

- **Datagram** This is a message that does not require a reply. An application may define that it requires a report on the status of the datagram.
- **Request** This is a message that requires a reply. The application has to provide the queue name (and the queue manager name) where to send the reply. Additionally, an application may define that it requires a report on the status of the datagram.
- **Reply** Reply to an earlier request message. The application that has to send the replay must either know where to send it or it has to retrieve the queue name (and the queue manager name) from the request.
- **Report** A status report of the message. An application sending a datagram message or a request has to define what reports it requires. An application retrieving a datagram message or a request has to check if it has to send reports.

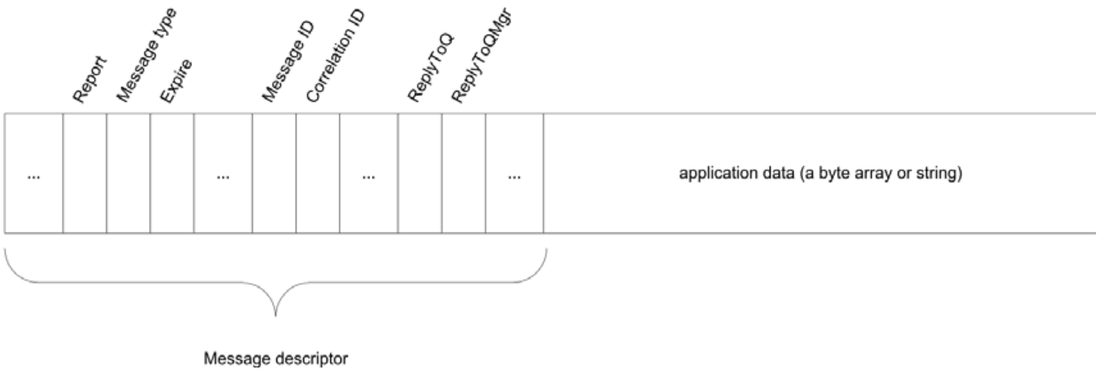
There are six types of reports:

Exception	WebSphere MQ sends an exception when a message cannot be delivered. An application can send an exception when an error occurs while performing an action triggered by a message.
Expiration	WebSphere MQ sends this type of report if the message is discarded prior to delivery to an application because its expiry time has passed.
Confirm on arrival (COA)	WebSphere MQ sends a COA when a message is put into the destination queue.
Confirm on delivery (COD)	WebSphere MQ sends a COD when an application read the message from the destination queue.

Positive action notification (PAN)	An application sends a PAN when it accepts a message.
Negative action notification (NAN)	An application sends a NAN when it does not accept a message.

The report types are described in detail in the IBM WebSphere MQ documentation.

A message consists of control data in the message descriptor and the application data. The application data is a character array or a string. The message descriptor describes the type of message and how to handle the message.



MQ-Interface currently only promotes some of the parameters to the public. These parameters can be used by an application to control messages. The remaining parameters are direct managed by MQ-Interface.

- The Report parameter defined what reports an application want to receive.
- The Message type specifies a datagram, request, reply or report message type.
- The Expire parameter defines the time how long a message is kept in a queue.
- The Message ID parameter is defined by the application. It may be used to define the action that will be performed when an application reads the message from the queue.
- The Correlation ID parameter is defined by the application. It may be used to define the action that will be performed when an application reads the message from the queue.

- The ReplyToQ parameter defines the queue where an application has to send a reply or request.
- The ReplyToQMgr parameter defines the queue manager where an application has to send a reply or request.

Message Exchange Scenarios

These scenarios describe the way an application can use WebSphere MQ to send and receive messages to and from other applications.

MQ-Interface supports two sets of scenarios. One set of scenarios covers incoming requests and asynchronous messages. These scenarios are handled in a separate process that runs an infinite loop and waits for messages. The other set of scenarios covers the message exchange between one application and another application (client-server communication). This set of scenarios can be handled by a message handler in the application that are directly linked to the MQ-Interface. The MQ-Interface can be direct linked since we assume that an application cannot perform another task while it waits for a reply to a request it has send to another application.

Incoming Messages

A VisualWorks MQ-Interface process listens on the queue for incoming messages. A message handler runs an infinite loop in a separate process where the process performs a GET with infinite wait interval. If data arrives, the handler identifies the action from the message and performs the appropriate method with the data of the message. Then it waits again for incoming messages.

Incoming asynchronous messages

In this scenario an application sends an asynchronous message to the receiver application. The sender does not want any response. The listener identifies the action to be appropriate method. The application does not send any message back to the sender, not even an error report.

1. The handler waits (GET with a infinite wait interval) for incoming messages.
2. The handler identifies the action (to be defined).
3. The handler performs the appropriate method.

4 The handler waits (GET with a infinite wait interval) for incoming messages.

Incoming asynchronous message with report

An application sends an asynchronous message to an application. The application has to inform the sender on defined states of the action triggered by the message. There are three statuses a sender may require:

- The action failed and generated an error
- The action was performed unsuccessfully or the component refuse to perform the action
- The action was performed successfully

The sender specifies which status reports it wants to receive from the application. There are other statuses a sender may require, but they are handled by WebSphere MQ itself (like notification on delivery).

1 The handler waits (GET with a infinite wait interval) for incoming messages.

2 The handler identifies the action (to be defined).

3 The handler performs the appropriate method.

4 The target performs the method.

5 The target identifies the type of information it has to report to the sender (parameter Report is set to either MQRO_EXCEPTION*, MQRO_PAN or MQRO_NAN).

6 If one of the events occur

   - the target creates a report message.
   - the target identifies the queue where to send the report (parameter ReplyToQ and ReplyToQMgr).
   - the target send the report to the queue.

7 The handler waits (GET with a infinite wait interval) for incoming messages.

Incoming Request

An applicationsends a request to the application. The application has to send a reply back to the sender when the action triggered by the request is finished. The application will always send a reply. If an error occurs then the error will be send back as reply.

1. The handler waits (GET with a infinite wait interval) for incoming messages.
2. The handler identifies the action (to be defined).
3. The handler performs the appropriate method.
4. The target performs the method.
5. If the action is performed successfully
   - the target generates a reply containing the result of the execution of the method for the request,
   - the target identifies the queue where to send the reply (parameter ReplyToQ and ReplyToQMgr),
   - the target sends the reply to the queue.
6. If the action is not performed successfully
   - the target generates an reply containing the error for the request,
   - the target identifies the queue where to send the report (parameter ReplyToQ and ReplyToQMgr),
   - the target sends the reply to the queue.
7. The listener waits (GET with a infinite wait interval) for incoming messages

Client-Server Message Exchange

This section describes scenarios in which an application starts the communication. It sends a request or an asynchronous message to another application (a server). It may wait for a reply or report. The application controls the communication. We assume that if the application is waiting for a response no other task can be performed. Thus, the message handler in the application can be directly linked to the MQ-Interface and the handler—and with this the application—can be blocked while it waits for a reply or report.

We recommend that there be a message handler for each external application with which the application wants to exchange data. The message handler implements the protocol the application uses to communicate with a server. WebSphere MQ transports only raw data (a byte array). Thus, the handler must have functionality to marshal VisualWorks objects into a byte array and vice versa.

Send asynchronous message

A message handler in the application sends a asynchronous message to another application. The component does not require any response.

1 The component opens the queue that is assigned to the server to which the application wants to send the message (only if the queue is not already open).

2 The component generates a asynchronous message (parameter MsgType set to MQMT_DATAGRAM) for the data sent to the server.

3 The component PUT the message to the server queue.

Send asynchronous message waiting for a report

A message handler in the application sends an asynchronous message to another application. It wants to be informed if the action triggered by the messages was performed successfully or if an error occurred while performing the action. When using this scenario, the application should always define at least a report for a successful performed action and for an error report. Independent of the result of the action there should be a feedback.

1 The handler opens the queue that is assigned for the server to which the application wants to send the message (only if the queue is not already open).

2 The handler generates a asynchronous message (parameter MsgType set to MQMT_DATAGRAM) for the data to be sent to the server.

3 The handler defines the reports it want to receive (parameter Report set at least to MQRO_EXCEPTION* and MQRO_PAN).

4 The handler adds the name of its own receiver queue to the message (parameter ReplyToQ and ReplyToQMgr).

5 The handler PUT the message to the server queue.

6 The handler issue a GET with a defined wait interval on its own receiver queue.

7 If an error report arrives in time raise an exception.

8 If a PAN or NAN arrives process it.

9 In no message arrives in time raise an exception.

Send Request

A message handler in the application sends a request to another application (server) and waits for a reply from this application. If the action triggered by the request fails the server sends an error report to the application. If the action was performed successfully the server sends a reply to the application.

1. The handler opens the queue that is assigned for the server to which the application wants to send the message (only if the queue is not already open).
2. The handler generates a request (parameter MsgType set to MQMT_REQUEST) for the data to be sent to the server.
3. The handler defines the report it want to receive (parameter Report set to MQRO_EXCEPTION*).
4. The handler adds the name of its own receiver queue to the message (parameter ReplyToQ and ReplyToQMgr).
5. The handler PUT the request to the server queue.
6. The handler issue a GET with a defined wait interval on its own receiver queue.
7. If a reply containing an error arrives raise it.
8. If a reply containing a valid reply arrives process it.
9. In no message arrives in time raise an exception.

Message Handler

A message handler is a set of classes that represent an external application. The application exchanges messages with the application through the message handler. Each handler has two queues:

- A sender queue where it writes asynchronous messages and requests. The external application will read messages from this queue.
- A receiver queue where it read replies and reports. The replies and/or reports are generated by the external application as response to a request the application sent to the external application.

Since WebSphere MQ and the MQ-Interface only deal with raw data (a byte array) the message handler has to implement the marshaling of VisualWorks objects to a byte array and vice versa. Optionally, the message handler may have an agent that is doing the marshaling for it.

2

Using MQ-Interface

This chapter discusses the public classes and methods provided in the MQ-Interface library and ther use in building an application.

Public Classes

The following are the classes in the VisualWorks MQ-Interface that are intended for use by an application programmer to access the functionality of WebSphere MQ.

QueueManager

An abstract class that represents a WebSphere MQ queue manager. It instantiates a concrete class based on parameters an application provides in the instance creation methods.

LocalQueueManager

A concrete queue manager class that connects the VisualWorks image to a queue manager that is installed on the same machine as the image, using the WebSphere MQ non-threaded server library.

LocalThapiQueueManager

A concrete queue manager class that connects the VisualWorks image to a queue manager that is installed on the same machine as the image, using the WebSphere MQ threaded server library.

RemoteQueueManager

A concrete queue manager class that connects the VisualWorks image to a queue manager that is installed on a different machine, using the WebSphere MQ non-threaded client library

RemoteThapiQueueManager

A concrete queue manager class that connects the VisualWorks image to a queue manager that is installed on a different machine, using the WebSphere MQ threaded client library.

MessageQueue

An abstract class that represents the link to a queue in a queue manager. It links to a queue by opening it. In WebSphere MQ a queue can only be opened for writing or for reading.

ReceiverQueue

A concrete message queue class that is used for reading messages from a queue.

SenderQueue

A concrete message queue class that is used for writing messages to a queue.

MQMessage

An abstract class for the four different message types IBM defined for WebSphere MQ.

ActionMessage

An abstract class for messages that is indented to trigger an action in the receiver of the message.

Request

A concrete message class for a request. Use this message type if the receiver has to create a reply.

AsynchronousMessage

A concrete message class for a message that simply carry some information to another application. It does not generate a reply.

Reply

A concrete message class for the reply an application generates when receiving a request.

Report

A concrete message class for reports generated by WebSphere MQ itself or by an application about the state of a message or an error.

ActionDecorator

An abstract class to parameterize the storage of the action selector.

DefaultActionDecorator

A concrete action decorator class that combines the action selector with the application data.

Program Flow

An application should connect to the queue manager when it initializes its external resources. This is usually done when the application is starting. Once connected, the application is able to use the services provided by WebSphere MQ. It should disconnect from the queue manager when the application discards its external resources. This is usually when the application is terminated.

An application has to open a queue located in a queue manager before it can send or receive messages. It should close the queue when it is no longer needed. An application may send several messages to an open queue or receive several messages from an open queue. A typical life cycle in the MQ-Interface can follow the sequence below:

```
Connect to a queue manager
  Open a receiver queue
  Open a sender queue
    Send messages to the sender queue
    Receive messages from the receiver queue
  Close the sender queue
  Close the receiver queue
Disconnect from the queue manager
```

You may skip the closing of queues since the queue manager will handle the life cycle of its queues.

Queue Mangers

A queue manager is a kind of service provider for WebSphere MQ services. An application connects to a queue manager in order to use its services. Typically, an application connects to the queue manager when it initializes its external resources.

A queue manager can be installed on the same machine as the image (local queue manager) or on another machine (remote queue manager). For both setups IBM provide a different shared library (DLL on Windows). Additionally VisualWorks needs a different syntax for threaded and non-threaded API calls.

The MQ-Interface contains a four concrete queue manager classes:

LocalQueueManager	local queue manager	non-threaded
LocalThapiQueueManager	local queue manager	threaded
RemoteQueueManager	remote queue manager	non-threaded
RemoteThapiQueueManager	remote queue manager	threaded

An application connects to a queue manager by using the instance creation message new: *aSymbol* threaded: *aBoolean* of the class QueueManager. The parameters determine the concrete subclass. The first parameter defines the location of the queue manager: #local for a local queue manager or #remote for a remote queue manager. The second parameter defines if we use threaded (true) or non-threaded (false) API.

Additionally a queue manager serves as a container for its queues. The QueueManager instance keeps a list of all queues that are created in its focus. This allows the queue manager to maintain the life cycle of its queues. Each queue belongs to one queue manager but an application may send messages to queues in different queue managers.

Local Queue Manager

An instance of LocalQueueManager or LocalThapiQueueManager represents a queue manager that is installed on the same machine as the VisualWorks image. For this setup we only need to know the name of the queue manager.

```
| manager |
manager := QueueManager new: #local threaded: false.
manager name: 'venus.queue.manager'.
manager connect.
```

The above code fragment connects to a local queue manager using the non-threaded API. This queue manager is represented by the class LocalQueueManager. This will be a bit faster then using the threaded API but it will bock the whole image when calling a procedure in the library. You should only use a non-threaded manager for a client that requests services from a server and that will actively waiting for replies.

```
| manager |
manager := QueueManager new: #local threaded: true.
manager name: 'venus.queue.manager'.
manager connect.
```

By simply changing the second parameter of the instance creation method new:threaded: you can create a local queue manager using threaded API. This queue manager is represented by the class LocalThapiQueueManager. Every API call will only block the actual process and not the whole image. But there is a small overhead for the threaded calls. They are slower, but do not block.

When you no longer need to exchange information (i.e., you terminate the application) you send the message disconnect to the queue manager instance. This will close any open queue; you don't have to close each queue individually.

A queue manager has a registry in which it keeps track of all its message queues. When an application requests a message queue the first time it creates a new message queue instance and registers it. When the application requests an already registered queue, it simply returns the registered queue.

Remote queue manager

An instance of RemoteQueueManager or RemoteThapiQueueManager represents a queue manager that is installed a machine other than the on which VisualWorks image resides. The image connects to the queue manager through a channel. In additionto the name of the manager we have to know:

- the machine where the queue manager is installed,
- the port of the remote queue manager listener. An application has to provide this information if there are more than one queue manager installed on the host. If there is only one queue manager on the host then this setting can be omitted.
- the name of the channel that connect the machine where the VisualWorks image is installed to the queue manager.

The instance creation is similar to the instance creation of a local queue manager but you have to provide the connection parameter.

```
| manager |
manager := QueueManager new: #remote threaded: false.
manager name: 'venus.queue.manager'.
manager host: 'Fuji1'.
manager port: 1423.
manager channel: 'cei.snk99.srvconn'.
manager connect.
```

The above code fragment connects to a remote queue manager using the non-threaded API. This queue manager is represented by the class RemoteQueueManager. Remember that when using non-threaded libraries every API call will block the whole image.

```
| manager |
manager := QueueManager new: # remote threaded: true.
manager name: 'venus.queue.manager'.
manager host: 'Fuji1'.
manager port: 1423.
manager channel: 'cei.snk99.srvconn'.
manager connect.
```

By simply changing the second parameter of the instance creation method new:threaded: you can create a remote queue manager using the threaded API. This queue manager is represented by the class RemoteThapiQueueManager. Every API call will only block the actual process and not the whole image. But there is a small overhead for the threaded calls, so the processes are slower.

Message queues

A MessageQueue instance represents a queue that is or will be opened for either reading information from it or writing information to it. A message queue is identified by its name. When creating a new MessageQueue object it will not automatically open the real queue in the queue manager. The queue will be opened when the first get: or put: message is sent to the MessageQueue instance.

This enables an application developer to initialize all needed queues at start time without caring about external resources. The developer just has to create instances for all necessary queues.

An application may define a timeframe for keeping a queue open. If no get: or put: message is sent to the queue within this timeframe, then the queue instance will close the queue. The instance is now in the state it was just after it was created. By default this timeframe is infinite.

A queue cannot be opened for both reading and writing information. You have to define what you want to do with the queue. The functionality for reading information and for writing information is separated into two subclasses.

A queue can be closed by sending the message close to it. It will also be closed if the queue manager that contains the queue is terminated.

Receiver queue

A receiver queue is used to read messages from a queue. You create a ReceiverQueue instance by sending the message getReceiverQueue: to a QueueManager instance.

```
| receiver |
receiver := queueManager getReceiverQueue: 'armor.1.reply'.
receiver waitInterval: 60000.
```

Now you can receive messages from the queue. In the above code fragment we create a receiver queue with the name “armor.1.reply”. We set the default wait interval to 60000 milliseconds or 60 seconds. This default setting is later used for receiving messages. The queue will wait for 60 seconds after you sent a get message to the receiver queue. If no message arrives in time the queue will raise an exception. An application may override this setting every time it wants to receive a message.

Normally when retrieving a message the queue will return the first message in the queue. Using a match parameter, you can select a specific messages from the queue. You can select using the message id, orrelation id, group id, sequence number, offset or token. The MQ-Interface allows several setups to define match parameter. Using the class method defaultMatchingParameter: a developer can define match parameter for all queues. This setting can be overridden in a queue by defining individual parameter for a queue using the instance method matchingParameter:.

When issuing a GET the developer can define special match parameter for a single GET.

Sender queue

A sender queue is used to send messages to a queue, which then sends it on to another application. You create a SenderQueue instance by sending the message getSenderQueue: to a QueueManager instance.

```
| sender |
sender:= queueManager getSenderQueue: 'armor.1.ent.p2p'.
sender setReceiverQueue: receiver.
```

The above code fragment creates a new SenderQueue instance. This sender queue can now be used to send messages to another application.

The second statement links a receiver queue to the sender queue. When sending a request or an asynchronous message, the name of the receiver queue and the name of its queue manager will be copied to the message. The receiving application needs this information to send a reply or report back to the sending application.

An application may use this convenience feature to simplify the message handling. If you don’t want to use this feature, you have to provide the queue and queue manager name for requests and asynchronous messages. If you use this feature, you can override the queue and queue manager name for any message you send through sender queue.

Messages

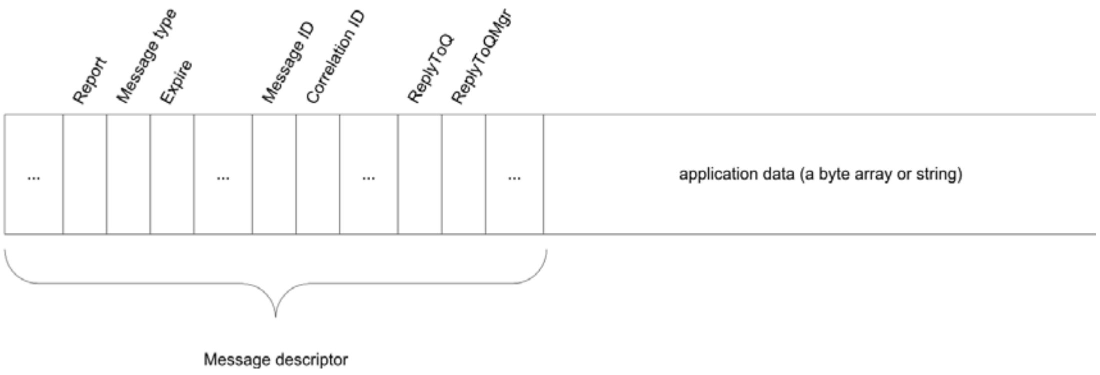
Messages represent the information applications exchange via the message queues. Messages only carry raw data. An application itself is responsible for marshaling objects to a byte array and to demarshaling a byte array to an object. This is not part of the MQ-Interface.

IBM defines four types of messages:

- Datagram - A message for which the application does not expect a reply. In the MQ-interface this message type is name asynchronous message since datagram normally has a different meaning.
- Request - A message for which the application expects a reply.
- Reply - A message that is a reply to a request.
- Report - A status or error message.

The MQ-Interface provides a class for each message type.

A message contains a description that defines how a message is to be handled by WebSphere MQ and the receiving application.



The MQ-Interface preset the parameters according to the use of the message and the settings of the queue. A developer may set some of the parameters to change the default settings and settings copied from the queue:.

Report

By default this parameter is not set. A developer can request a status or an error report by defining the type of the report. The MQ-Interface provides a set of methods to define the reports to be sent back to the sender queue. This parameter is only valid for requests and for asynchronous messages.

- reportError - Request an error report if a message cannot be delivered or if the action triggered by a message failed.
- reportCOA - Request a report when the message is placed into the destination queue.
- reportCOD - Request a report when the message is read from the destination queue.
- reportNAN - Request a report when the receiving application accepts the message.
- reportPAN - Request a report when the receiving application rejects the message.
- flushReports - Reset the report parameter.

MessageType

Defines the type of a message. Currently this are the four types defined by Websphere MQ.

Expire

This parameter defines how long a message will be kept in a queue. If nobody read the message from the queue within this time the queue manager will remove it from the queue. By default this is set to unlimited. Somebody else (i.e. Tivoli) has to take care.

Message ID

The message identifier. This is a string that distinguishes one message from another. By default WebSphere MQ will generate this parameter.

Correlation ID

This is a string that relates one message to another (i.e., a reply to a request). By default WebSphere MQ will generate this parameter.

ReplyToQ

This parameter defines the reply queue to which a reply or report message has to be sent. It is only set for requests and asynchronous messages. An application retrieving a request has to use this parameter to identify the queue where to send a reply.

ReplyToQmgr

This parameter defines the queue manager that contains the reply queue to which a reply or report message has to be sent. It is only set for requests and asynchronous messages. An application retrieving a request has to use this parameter to identify the queue manager where to send a reply.

Messages that trigger an action in the receiver must store a selector for the action somewhere inside the message itself. The selector can be coded into the application data or it can be stored into an unused parameter in the message descriptors. The MQ-Interface uses a decorator for defining the storage of the action selector. The MQ-interface has a variable that contains a decorator object. An application can set the decorator individually for each message. For convenience, the class MQMessage contains a class method where an application can define a decorator that should be used for all messages.

The messages action, action:, data, and data: call the same methods in the decorator. The methods in the decorator know how to store the action selector and how to retrieve it. The two methods below show the messages action and action: for a decorator that stores the action selector into the parameter applicationName of the message descriptor.

```
action
   "Answer the action selector coded into a parameter
   of the message descriptor"

   ^message applicationName

action: aString
   "Answer the action selector coded into a parameter
   of the message descriptor"

   message applicationName: aString
```

The messages data and data: implemented in the base class ActionDecorator just store the application data into the message. They support a decorator that stores the selector in the application data. The concrete subclass DefaultActionDecorator does this.

Asynchronous message

An asynchronous message represents a message that is used by an application to send information to another application but does not require a reply. An application may define reports to be sent back. A developer creates a new asynchronous message by sending the message newAsynchronousMessage to a sender queue.

```
| message|
message := sender newAsynchronousMessage.
message action: #tick.
message data: #[ 84].        "Integer value for T"
sender put: message.
```

The above code fragment sends a time tick with a Boolean value (true) to another application.

If we want to know if the message is delivered or if the other application accepts the message we have to define a report.

```
| message report |
message := sender newAsynchronousMessage.
message action: #processOrder.
message data: anOrder asMQData.
message reportCOD.
sender put: message.
report := receiver getReportFor: message.
```

This code fragment sends an request to the receiver application to let us know if the other application read the message from the queue.

The code fragment waits for a report after it has sent the message. The queue manager creates that report when the receiver application reads the message from the queue. In this example the default wait interval of the queue is used. If no report arrives within this time interval the queue raises an exception. If you want to use a different wait interval you have to use the message getReportFort:wait:. The second parameter of the message defines the wait interval.

In the above example we assume that the sender queue is linked to a receiver queue. The sender queue copies the name of the receiver queue and the name of the queue manager into the message before it send it. If the queues are not linked then the developer has to set the queue parameter manually as shown in the fragment below.

```
| message report |
message := sender newAsynchronousMessage.
message action: #newOrder.
message data: anOrder asMQData.
message reportCOD.
message replyQueue: receiver.
sender put: message.
report := receiver getReportFor: message.
```

The queue will create the message and correlation ids. After the message is sent the parameter can be read from the message description. The receiver now can associate a report it receives to the message. If you want to define your own message and correlation id you can do so by using the messages messageID: and correletionID:.

You may define several reports. Based on the report you define you may have to read several reports. The reports will arrive in the following order:

COA	Confirmation of arrival	the message is place into the destination queue
COD	Confirmation of delivery	the message is read from the destination queue
PAN/NAN	Positive or negative action notification	the retrieving application accepts or rejects the message. They should always used together.

An error report may arrive anytime. The queue manager creates an error report when the message cannot be delivered or an application send an error report if the action triggered by the message failed.

If you define all report types you have to retrieve three reports: COA, COD, PAN/NAN. You only break this if you retrieve an error report.

Request

Use a request if you want to receive a response to a message you sent to another application. An application that retrieves a request has to create a reply and send it back to the sender application.

```
| request reply |
request := sender newRequest.
request action: #saveAddress.
request data: anAddress asMQData.
sender put: request.
reply := receiver getReplyFor: request.
```

The above code fragment creates a request to store an address into a server. The server will process the request and send a reply back to the sender. In the example we assume that the sender and receiver queues are linked – the sender knows the queue where the request will arrive. The sender copies the name of the queue and the name of the queue’s queue manager into the message. Since we do not define a message or correlation id WebSphere is doing it for us.

You can define the same reports for a request as for an asynchronous message. They will be handled the same as described above. The following example shows a code fragment where we want to know if the message is delivered.

```
| request reply report |
request := sender newRequest.
request action: #saveAddress.
request data: anAddress asMQData.
message reportCOD.
sender put: request.
report := receiver getReportFor: request.
reply := receiver getReplyFor: request.
```

The queue manager creates the report when the receiver read the request from the queue. The application process the message – stores the address – and sends a reply back to the same queue as the queue manager sent the report. Thus, the report will arrive before the reply and we have to retrieve them in this order.

Reply

If an application retrieves a request it must send a reply back to the queue defined in the request.

```
listenForIncommingMessages
"A simple WebSphere MQ listener"

| message reply answer |
[message := listener getIncommingMessage.
sender := self queueManager
getSenderQueue: aMessage replyQueueName.
answer := self
   processAction: message action
   with: message data.
reply := message createReply.
reply data: answer asMQData.
sender put: reply.
true] whileTrue: [].
```

This sample method can be use as a simple listener that route any incoming message to its own class. The class process the action coded into the request. The listener then creates a reply, copies the response into the reply and sends it back.

A request has to carry information – the name of the receiver queue and the name of its queue manager - where to send the reply. The receiving application uses this information to send the request back to the sending application. The message createReply will copy all necessary parameter from the request to the reply.

In this example we assume that we only have one queue manager. Thus we don’t have to handle the queue manager parameter.

This message can be forked – maybe in an initialize - as a separate process running in the background. Then it does not block the rest of the image.

```
initialize
   "Fork a process for a listener"

   listener := self queueManager
   getReceiverQueue: 'app1.rec.queue'.
   process := [self listenForIncommingMessages].
   process forkAt: Processor userInterruptPriority.
```

Report

An application may define reports it want to receive when sending an asynchronous message or a request to another application. Some of the reports must be created by the receiving application.

```
listenForIncommingMessages
"A simple WebSphere MQ listener"

| message reply report answer |
[message := listener getIncommingMessage.
sender := self queueManager
getSenderQueue: aMessage replyQueueName.
(message requiresPANReport and: [self accept: message action])
ifTrue: [sender put: (message createReport: #PAN)].

answer := self
   processAction: message action
   with: message data.
reply := message createReply.
reply data: answer asMQData.
sender put: reply.
true] whileTrue: [].
```

The above sample work the same as the example before except it checks if it has to send a PAN back to the sending application before it process the action triggered by the message. The sample sends the report through the same queue as the reply. The sending application first has to retrieve the report and then it has to retrieve the reply.

The message `createReport:` creates a new report and copies all necessary parameter to the report. The parameter defines what report is send back. Currently we support three report types:

- PAN is sent when the application accepts the message.
- NAN is sent when the application rejects the message.
- Error is sent when the action triggered by the message raised an error.

3

Class Reference

This chapter describes all classes and public methods of the MQ-interface.

QueueManager

QueueManager is an abstract class that represents a WebSphere MQ queue manager. It instantiates a concrete class based on parameters in the instance creation methods. A queue manager keeps two lists of all queues an application requests. One list for receiver queues and one list for sender queues. The manager uses these two lists to maintain the life cycle of its queues.

Super class

Object

Class methods

new: *aSymbol* **threaded:** *aBoolean*

This factory method create a concrete subclass of QueueManager based on the parameter of the message. WebSphere MQ supports two different setups for the queue manager. Additionally, it provides libraries for threaded and non-threaded API calls. Therefore there are four subclasses that are defined by the two parameter of the message.

aSymbol	aBoolean	
#local	true	LocalThapiQueueManager
#local	false	LocalQueueManager
#remote	true	RemoteThapiQueueManager
#remote	false	RemoteQueueManager

Instance methods

name

Return the name of the queue manager

name: *aString*

Set the name of the queue manager

connect

Connects an instance of a concrete subclass to the WebSphere MQ queue manager. The application now can use other WebSphere MQ services.

disconnect

Disconnect the application from the WebSphere MQ queue manager. The application cannot use WebSphere MQ services anymore.

getReceiverQueue: *aString*

Request a ReceiverQueue instance for a queue name. If the queue manager has already a receiver queue registered for this name it just answer the queue. If it does not have a receiver queue registered for it, it will create one and register it.

getSenderQueue: *aString*

Request a SenderQueue instance for a queue name. If the queue manager has already a sender queue registered for this name it just answer the queue. If it does not have a sender queue registered for it, it will create one and register it.

unregisterQueue: *aMessageQueue*

Discard the given queue from the internal queue cache. This is useful in connection with dynamic caches, which can accumulate in the QueueManager cache. To close and unregister a queue, use closeAndUnregisterQueue: *aMessageQueue*.

LocalQueueManager

LocalQueueManager is a concrete queue manager class that connects a VisualWorks image to a queue manager that is installed on the same machine as the image. Ituses the WebSphere MQ non-threaded server library. This class does not define its own public protocol. It just defines the DLLCC class for the threaded server library.

Super class

QueueManager

LocalThapiQueueManager

LocalThapiQueueManager is a concrete queue manager class that connects a VisualWorks image to a queue manager that is installed on the same machine as the image. It is using the WebSphere MQ threaded server library. This class does not define its own public protocol. It just defines the DLLCC class for the threaded server library.

Super class

LocalQueueManager

RemoteQueueManager

RemoteQueueManager is a concrete queue manager class that connects a VisualWorks image to a queue manager that is installed on a different machine using the WebSphere MQ non-threaded client library.

Super class

QueueManager

Instance methods

host

Returns the name or the ip-address of the host where the queue manager is installed.

host: *aString*

Set the name or the ip-address of the host where the queue manager is installed. When using the host name to define the host then the local machine must be able to resolve the ip-address for the host name.

port

Returns the port of the listener.

port: *aNumber*

Set the port of the listener. This must only be set if there are more then one queue manager installed on the remote server.

channel

Return the name of the channel that connects the application to the remote queue manager.

channel: *aString*

Set the name of the channel that connects the application to the remote queue manager.

RemoteThapiQueueManager

A concrete queue manager class that connects a VisualWorks image to a queue manager that is installed on a different machine using the WebSphere MQ threaded client library. This class does not define its own public protocol. It just defines the DLLCC class for the threaded client library.

Super class

RemoteQueueManager

MessageQueue

MessageQueue is an abstract class that represents the link to a queue in a queue manager. In WebSphere MQ a queue can only be opened for writing or for reading. It is not possible to open a queue for both. Therefore the MQ-Interface contains two subclasses of this class that represent the different behavior for sender and receiver queues.

Initially an instance of a MessageQueue subclass is not linked—by opening—to its real WebSphere MQ queue. The object opens the queue when the first get: or put: message is sent to the object. A developer may force a queue to establish the link by sending the message open to it. A developer may define a timeframe after which a queue will be closed if no get: or put: message is sent to the queue within this timeframe.

Super class

Object

Instance methods

name

Answers the name of the queue

open

Opens a WebSphere MQ queue. Now an application can retrieve messages from it or send messages to it. This method is automatically called when an application sends the first get: or put: message to it.

close

Close the WebSphere MQ queue. An application may send this message manually. It also can leave it to the queue manager that maintains the live cycle of its queues.

timeout: *anInteger*

Define the time interval for closing the queue resource. If no get: or put: is issued in this frame the queue is closed but the instance remains in the same state as being just created.

ReceiverQueue

ReceiverQueue is a concrete message queue class that is or will be opened for reading messages from a queue.

Super class

MessageQueue

Class methods

defaultMatchingParameter: *aStringCollection*

Define the default parameters that will be used to retrieve selected messages from the queue. By default there are none. The first message in the queue is received from the queue.

Instance methods

matchingParameter: *aStringCollection*

Define the parameters that will be used to retrieve the selected messages from the queue. If this message is not sent, then the default parameters will be used to read messages from the queue.

getReportFor: *aMessage*

Try to receive a report from the queue. This method uses the default wait interval of the queue. If no report arrives in time then the queue raise an error. If a matching report arrives the method answers this report.

getReportFor: *aMessage* **match:** *aStringCollection*

Retrieve the first report from the queue that match the parameters provided by *aStringCollection*. This method overrides previous match settings in the queue.

getReportFor: *aMessage* **wait:** *aNumber*

Try to receive a report from the queue. This method uses a defined wait interval. The default value of the queue is ignored. If no report arrives in time then the queue raise an error. If a matching report arrives, the method answers this report.

getReportFor: *aMessage* **wait:** *anInteger* **match:** *aStringCollection*

Retrieve the first report from the queue that match the parameters provided by *aStringCollection*. The report is sent back by another application to an asynchronous message or request—the parameter of this method. This method is used to override the timeout and match settings of the queue.

getReplyFor: *aRequest*

Try to receive a reply from the queue. This method uses the default wait interval of the queue. If no reply arrives in time, then the queue raises an error. If a matching reply arrives, the method answers this reply.

getReplyFor: *aMessage* **match:** *aStringCollection*

Retrieve the first reply from the queue that matches the parameter provided by the last parameter. The reply is sent back by the application receiving the request. This method overrides previous match settings in the queue.

getReplyFor: *aRequest* **wait:** *aNumber*

Try to receive a reply from the queue. This method uses a defined wait interval. The default value of the queue is ignored. If no reply arrives in time then the queue raises an error. If a matching reply arrives, the method answers this reply.

getReportFor: *aMessage* **wait:** *anInteger* **match:** *aStringCollection*

Retrieve the first reply in the queue that matches the parameters provided by *aStringCollection*. The reply is sent back by the application receiving the request. This method is used to override the timeout and match settings of the queue.

getIncommingMessage

Read any message from the queue. This method uses the default wait interval of the queue. If no message arrives in time then the queue raise an error. If a matching message arrives the method answers this message. We recommend to define an infinitive wait and to use this method in a separate process to avoid the blocking of the whole image.

The main purpose of this method is in implementing a listener that handles unexpected incoming messages.

waitInterval: *aNumber*

Define the default wait interval for an attempt to retrieve a message from a receiver queue. The parameter defines the wait interval in milliseconds.

waitForever

Define the default wait interval for an attempt to retrieve a message from a receiver queue to infinite. A get: message will wait forever.

noWait

Define the default wait interval for an attempt to retrieve a message from a receiver queue to zero. A get: message will return immediately. If there is no message in the queue it will raise an error.

SenderQueue

SenderQueue is a concrete message queue class that is or will be opened for writing messages to a queue. A sender queue can be linked with a receiver queue. Asynchronous messages and requests sent to another application through the sender queue carry the name of the linked receiver queue and the name of its queue manager. Reports and replies will later arrive in the linked receiver queue.

Super class

MessageQueue

Instance methods

newAsynchronousMessage

Create a new asynchronous message and preset it according to the queue's settings. The message should be sent through the queue that created it.

newRequest

Create a new request and preset it according to the queue's settings. The message should be sent through the queue that created it.

receiverQueue: *aReceiverQueue*

Links a receiver queue to the sender queue. The sender queue will use this setting to set the queue name and queue manager name in messages sent through this queue.

put: *aMessage*

Write a message to the queue.

MQMessage

An abstract class for the four different message types IBM defined for WebSphere MQ.

Super class

Object

Instance methods

data

Answer the application data transferred by a message. WebSphere MQ only transfers raw data. An application is responsible to convert the byte array into its domain objects.

data: *aByteArray*

Set the application data transferred by the message. WebSphere MQ only transfers raw data, so the application is responsible to convert its domain objects into a byte array.

reportCOA

Set the flag for receiving a confirmation of arrival. This report is generated by the queue manager when the message is placed into the destination queue.

reportCOD

Set the flag for receiving a confirmation of delivery. This report is generated by the queue manager when the message is read from the destination queue.

flushReports

Reset the report flags. No report is expected.

replyQueue: *aQueue*

Define the queue where a receiving application has to send replies and reports. The method will copy the queue name and its queue manager name to the message.

ActionMessage

ActionMessage is an abstract class for messages that are intended to trigger an action in the receiver of the message. These message types carry a symbol specifying the action in one of the parameters of the message descriptor.

Super class

Object

Class methods

defaultDecoratorClass: *aClass*

Define the class of the default decorator that will be used to store and retrieve the action selector into the message.

Instance methods

actionDecorator: *aDecorator*

Set the decorator that codes the action message into the descriptor, into the application data or both.

action

Answers a string that identifies the action an application has to perform when retrieving an action message.

action: *aSymbol*

Set the action to be performed when receiving an action message.

reportError

Set a flag for receiving error reports. These reports are created by the queue manager when a message cannot be delivered or by an application if the action triggered by a message failed.

reportNAN

Set the flag for receiving a negative action notification. This report is generated by the receiver application when the message is rejected.

reportPAN

Set the flag for receiving a positive action notification. This report is generated by the receiver application when the message is accepted.

requiresErrorReport

Answer true if the receiver wants to receive error reports, answer false otherwise.

requiresPANReport

Answer true if the receiver wants to receive a positive action notification, answer false otherwise.

requiresNANReport

Answer true if the receiver wants to receive a negative action notification, answer false otherwise.

replyQueueName

Answer the name of the queue where a receiving application has to send replies and reports.

replyQueueManagerName

Answer the name of the queue manager that contains the queue where a receiving application has to send replies and reports.

createReport: *anActionMessage*

Create an Report instance for an action message. This method copies all necessary parameter from the action message to the report.

AsynchronousMessage

An AsynchronousMessage is a concrete message class for a message that simply carries some information to another application. It does not generate a reply. This class does not define its own public protocol.

Super class

ActionMessage

Request

Request is a concrete action message class for a request message. You should use this message type if you want to receive a reply to a message.

Super class

ActionRequest

Instance methods

createReply: *anActionMessage*

Create a Reply instance for an action message. This method copies all necessary parameter from the action message to the reply. The application only has to add the data to the reply.

Reply

Reply is a concrete message class for a reply an application has to generate when receiving a request. This class does not define its own public protocol

Super class

MQMessage

Report

Report s a concrete message class for reports generated by WebSphere MQ itself or by an application about the state of a message or an error.

Super class

MQMessage

Instance methods

isError

Answers true if it is an error report. Answers false otherwise.

isCOA

Answers true if it is an confirmation of arrival report. Answers false otherwise.

isCOD

Answers true if it is an confirmation of delivery report. Answers false otherwise.

isPAN

Answers true if it is a positive action notification report. Answers false otherwise.

isNAN

Answers true if it is a negative action notification report. Answers false otherwise.

ActionDecorator

ActionDecorator is an abstract class that stores the action selector somewhere in the message. A subclass has to implement the messages action and action: that implement the concrete storage of the action selector (i.e., in an unused parameter of the message selector).

Super class

Object

Instance methods

message: *aMQMessage*

Sets the message where the decorator stores and retrieves the action selector. This method is called when a decorator is assigned to a message.

data

Answers the data part of the message.

data: *aByteArray*

Copy the byte array into the data part of the message.

DefaultActionDecorator

DefaultActionDecorator is a concrete class that stores the action selector together with the application data in the data part of the message. The selector is stored before the application data. The first byte of the data part contains the size of the selector. This makes it easy to split the selector and the application data when an application receives the message. The class contains instance variables that cache the selector and application data. If both variables are set, then the decorator combines them and stores it as data part of the message.

Super class

Object

Instance methods

action

Answers the action selector of the message

action: *aString*

Sets the action selector of the message. If the data is set, too then combine them and store them in the data part of the message.

data

Answers the application data of the message.

data: *aString*

Sets the application data of the message. If the action is set, too then combine them and store them in the data part of the message.

Index