webMethods Broker Client Java API Programmer's Guide

Version 9.6

April 2014

WEBMETHODS

This document applies to webMethods Broker Version 9.6 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: BR-CLIENTJAVAAPI-PG-96-20161018

Table of Contents

About this Guide

The *webMethods Broker Client Java API Programmer's Guide* describes the application programming interfaces (API) that you use to create event-based applications with the Java language.

This document is intended for use by programmers who are developing event-based applications using webMethods Broker. The reader is assumed to possess general knowledge of programming and specific knowledge of the Java programming language.

This book assumes you are familiar with the terminology and basic operations of your operating system (OS). If you are not, please refer to the appropriate documentation for that OS.

Important: If you have a lower fix level installed, some of the features described in this document might not be available to you. For a cumulative list of fixes and features, see the latest fix readme on the Empower website at https://empower.softwareag.com.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies storage locations for services on webMethods Integration Server, using the convention *folder.subfolder:service* .
UPPERCASE	Identifies keyboard keys. Keys you must press simultaneously are joined with a plus sign (+).
Italic	Identifies variables for which you must supply values specific to your own situation or environment. Identifies new terms the first time they occur in the text.
`Monospace font`	Identifies text you must type or messages displayed by the system.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.

Convention	Description
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Documentation Installation

You can download the product documentation using the Software AG Installer. The documentation is downloaded to a central directory named _documentation in the main installation directory (SoftwareAG by default).

Online Information

You can find additional information about Software AG products at the locations listed below.

If you want to...	Go to...
Access the latest version of product documentation.	**Software AG Documentation website** http://documentation.softwareag.com
Find information about product releases and tools that you can use to resolve problems. See the Knowledge Center to: ■ Read technical articles and papers. ■ Download fixes and service packs (9.0 SP1 and earlier). ■ Learn about critical alerts. See the Products area to: ■ Download products.	**Empower Product Support website** https://empower.softwareag.com

If you want to...	Go to...
■ Download certified samples. ■ Get information about product availability. ■ Access older versions of product documentation. ■ Submit feature/enhancement requests.	
■ Access additional articles, demos, and tutorials. ■ Obtain technical information, useful resources, and online discussion forums, moderated by Software AG professionals, to help you do more with Software AG technology. ■ Use the online discussion forums to exchange best practices and chat with other experts. ■ Expand your knowledge about product documentation, code samples, articles, online seminars, and tutorials. ■ Link to external websites that discuss open standards and many web technology topics. ■ See how other customers are streamlining their operations with technology from Software AG.	**Software AG Developer Community for webMethods** http://communities.softwareag.com/

1 Getting Started

Overview

This chapter describes the basic features of the Java language API for the webMethods Broker system, and how to use it to build event-based client programs that communicate with each other and with other applications through Broker.

Event-Based Applications

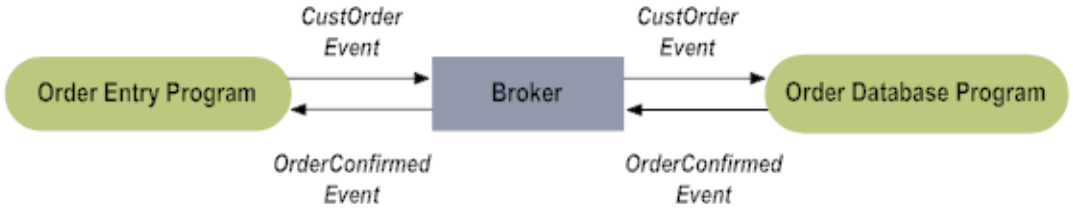
The webMethods Broker system provides you with interfaces, classes, and methods for building powerful, event-based applications that are made up of de-coupled *client programs*. The client programs are de-coupled because they communicate by sending and receiving *events* through a third entity called a Broker. A Broker is actually part of a Broker Server, which can contain multiple Brokers. For more information on the Broker Server, see *Administering webMethods Broker*.

The figure below depicts a simple example where an Order Entry program receives input from a customer and *publishes* a CustOrder event. The act of publishing the event causes the event to be sent to the Broker. The Broker then determines that the Order Database program has *subscribed* to the CustOrder event, and so the Broker distributes the event to the Order Database client program.

Sending events through a Broker

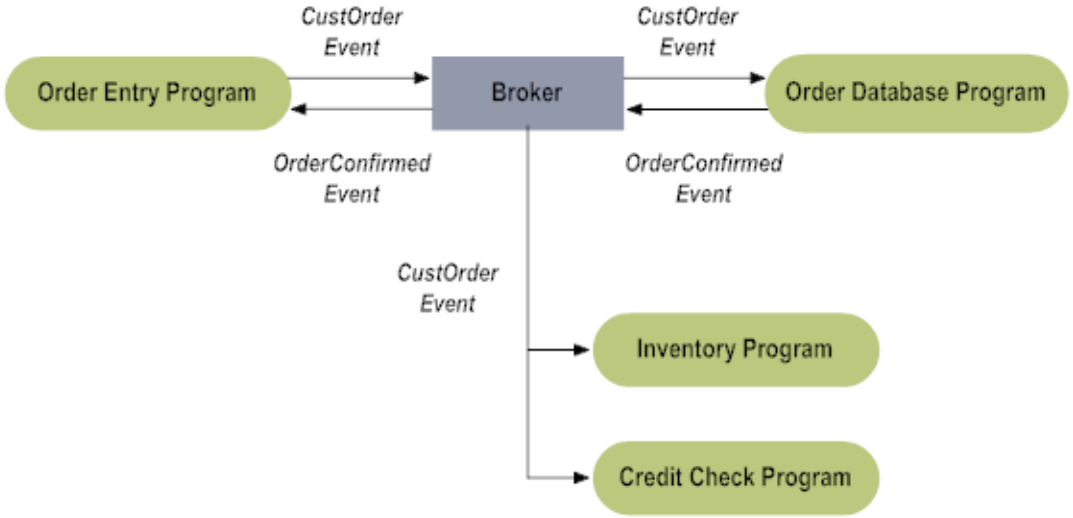
More complex applications are possible than the simple example shown in the next figure. The Order Database program could generate an OrderConfirmed event that would be sent to the Order Entry program as a reply to a CustOrder event.

Two clients implementing a request-reply event mode



You might also design a system where several client programs receive the CustOrder event and perform their own processing, such as checking the inventory on hand or verifying the customer's credit status.

Multiple clients subscribing to an event type



The webMethods Broker system also allows the Broker and the client programs that comprise an application to execute on different hosts within a network.

Events

Events are one of the most important ingredients in an application developed with the webMethods Broker system. Events are defined using Software AG Designer, described in *Software AG Designer Online Help*.

Note: Please note that Broker events are generally called *documents* in Designer and other webMethods components.

Events have these important characteristics:

- Events are organized into *event scopes*, allowing you to group the events related to your application.
- Events have a unique event type, which defines the *event fields* they contain and the data type associated with each field.
- Event fields represent data in a platform-independent representation, allowing clients on different platforms to exchange information.

Events are discussed in greater detail in "Creating and Initializing Events" on page 55.

Broker Clients

Client programs can consist of one or more Broker*clients*. Just as a program can open and use more than one file, a program can create and use multiple Broker clients. After created, a Broker client represents a connection to a particular Broker on a particular

host. A Broker client can subscribe to events, publish events, and receive events. Broker clients can also share a connection to a Broker and they can also share the same client queue and client state. Broker clients are covered in greater detail in "Using Broker Clients" on page 23.

The Broker

The webMethods Broker coordinates the exchange of events between client programs. To accomplish this complex task, the Broker:

- Queues all events that are published by Broker clients.
- Sends events to those clients which have subscribed to and are ready to receive the event.
- Provides various levels of reliable delivery of events through the use of event sequence numbers. Sequence numbers are described in "Using Sequence Numbers" on page 133.
- Provides event filtering services that allow Broker clients to selectively filter the events they will receive, based on event content. Filters are discussed in "Using Event Filters" on page 145.
- Maintains information on all event type definitions, which are defined with Designer. See the *Software AG Designer Online Help* for more information.

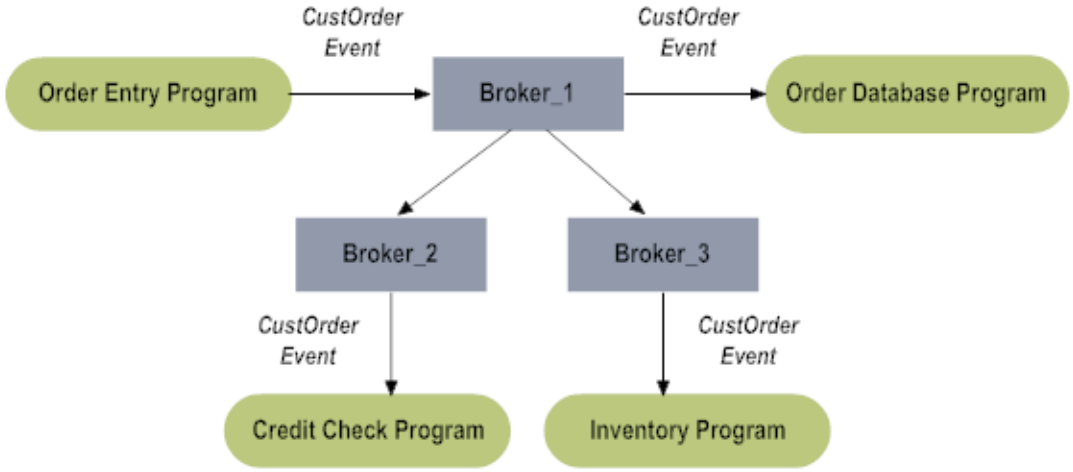
  Note: Broker events are known as Broker*document types* in Designer.

- Maintains information about *client groups*, which define event publication permissions, event subscription permissions, network access control, and client event queue characteristics for Broker clients in each group. See *Administering webMethods Broker* for details on client groups.
- Maintains an event queue and client state information for each Broker client that has been created.
- Maintains a dead letter queue, in which it retains events for which it has no subscribers.

Broker-to-Broker Communication

The webMethods Broker systems allows two or more Brokers to share information about their event type definitions and client groups. This sharing of information enables communication between Broker clients connected to different Brokers. The next figure shows how an event published by a client program connected to Broker_1 can be received by a client program connected to either Broker_1, Broker_2, or Broker_3.

Broker-to-Broker communication



In order to share information and forward events, Brokers must join a *territory*. All Brokers within the same territory have knowledge of one another's event type definitions and client groups. For more information on this feature, see *Administering webMethods Broker*.

Using the webMethods Broker Java API

The webMethods Broker Java API is implemented as a Java package that provides all the necessary interfaces, classes and methods for:

- Creating and manipulating events.
- Transferring events to a Broker.
- Retrieving events from a Broker.
- Querying the Broker for state information.

You can use the webMethods Broker Java API to create a wide variety of applications, including simple clients, watchdog agents, and adapters for legacy applications and existing data sources.

Application Development with webMethods Broker

Using the webMethods Broker system to develop applications involves the following steps:

1. Install the webMethods Broker system. See *Installing webMethods and Intelligent Business Operations Products* or your system administrator.
2. Start the Broker. See *Installing webMethods and Intelligent Business Operations Products* or your system administrator.

3. Design the client programs and the events that will comprise your system.
4. Use Designer to define the events for your application. See the *Software AG Designer Online Help* or your system administrator.

   Note: Broker events are known as Broker*document types* in Designer.

5. Ensure that your applications can locate the webMethods Broker Java package. You can do this by using one of the following techniques:
   a. Compile your application with the -classpath complier flag set to the location of the webMethods Broker Java package. *This is the recommended technique*.
   b. Install the webMethods Broker Java package in the same directory where you develop your applications. You must also ensure that your `CLASSPATH` environment variable includes the current directory specification.
   c. Add the location of the webMethods Broker Java package to your `CLASSPATH` environment variable. This is *not recommended* due to potential name conflicts with other Java packages that might be installed on your system.
6. Write and compile your client code.
7. Execute your client programs.

Application Deployment

When deploying your Java client applications, you must also deploy the necessary webMethods Broker classes. The webMethods Broker installation contains an archive with all of the files necessary for application deployment.

Note: This file must be in the application's `CLASSPATH` when the application is deployed.

Platform	Location
AIX, HP-UX, Linux, and Solaris	■ *Software AG_directory*/common/lib/wm-brokerclient.jar ■ *Software AG_directory*/common/lib/wm-g11nutils.jar
Windows	■ *Software AG_directory*\common\lib\wm-brokerclient.jar ■ *Software AG_directory*\common\lib\wm-g11nutils.jar

webMethods Broker Java API Online Documentation

The webMethods Broker Java API provides API-level documentation. The documentation is available as HTML web pages, in the following platform-specific location.

Platform	Location of webMethods Broker Java Packages
AIX, HP-UX, Linux, and Solaris	*Software AG_directory*/broker/doc/java_api/index.html
Windows	*Software AG_directory*\broker\doc\java_api\index.html

2 Using Broker Clients

Overview

This chapter describes the webMethods Broker Java API for creating, using, and destroying Broker clients in your applications. Reading this chapter should help you understand:

- How to create and use a BrokerClient object.
- The context associated with a BrokerClient, including client identifiers, client state, client groups, client life cycle, event queue storage types, and client infoset.
- How to disconnect and reconnect a BrokerClient.
- How to register callback methods for connection notification.
- BrokerConnectionDescriptor objects and their role in the sharing of client state and client connections between multiple Broker clients.
- How to use the BrokerConnectionDescriptor to enable automatic re-connection.
- How to use the BrokerConnectionDescriptor to enable the redelivery counting feature.
- How to use the BrokerConnectionDescriptor to enable the keep-alive feature.

Understanding Broker Clients

A client program creates one or more Broker clients in order to *publish* or retrieve events. For example, a network monitoring application might create a Broker client to publish events that represent network transmission errors. A network management application might create a Broker client to *subscribe* to these network error events. If the number of network error events retrieved reaches a critical threshold, the management application might create a different Broker client and use it to publish statistics about the network failure.

Event-based client applications are de-coupled from one another because they generate and receive events through an entity called a Broker Server. When your client program creates a Broker client, it is actually establishing a connection between your application and a Broker running on the local host or some host on the network.

Client State

When a connection is established between your Broker client and a Broker, the Broker creates and maintains a *client state* that includes the following information:

- A client identifier that uniquely identifies your Broker client to a particular Broker.
- A queue where events received for your Broker client will be stored until they are retrieved.

- A list of event subscriptions for your Broker client.
- The *client group* with which your Broker client is associated.

A single client state can be shared by multiple Broker clients, as described in "Sharing Client State" on page 40.

Client Groups

Every Broker client belongs to a *client group*, which provides important security features by limiting the behavior of the member Broker clients. Client groups are defined and maintained by your webMethods Broker administrator. Each client group is given a name. The restrictions on client group names are discussed in "Parameter Naming Rules" on page 199.

Client groups define these important limits:

- The event types that be published or delivered by the group members.
- The event types that be delivered to group members and for which they register subscriptions. See "Event Subscriptions" on page 80 for more information.
- The client *life cycle*, or how long the Broker will maintain client state information for each group member.
- The client queue type, which determines how the events are stored by the Broker.
- Security information that determines which entities create a Broker client in this group.

Client Life Cycle

The *life cycle* associated with your Broker client's client group will determine how long the Broker will maintain the client state for your Broker client.

If the life cycle is *explicit-destroy*, the client state can only be destroyed by a system administrator or by your client application calling the BrokerClient.destroy method. The explicit-destroy life cycle is useful for client applications that need to maintain state information even if a network or system failure occurs.

If the life cycle is *destroy-on-disconnect*, the Broker will destroy the client state whenever the connection between the Broker client and the Broker is lost. The destroy-on-disconnect life cycle is used by client applications that do not need to maintain state information in the event of a network or system failure.

Queue Storage Types

The client *queue storage type*, also defined by your Broker client's client group, can be either *volatile* or *guaranteed*.

Queue Type	Description
volatile	The Broker will queue events for your Broker client using local memory. This offers higher performance along with a greater risk of losing events if a hardware, software, or network failure occurs.
guaranteed	The Broker will use a two-phase commit process to queue events for your Broker client. This offers the lowest performance, but very little risk of losing events if a hardware, software, or network failure occurs.

Note: Storage types also be defined for a particular event type. See "Obtaining Storage Type Property" on page 141 for information on how client group and event type storage specifications interact.

Client Infoset

Each BrokerClient is allowed to store information about its state or configuration in a single client infoset. The BrokerClient.getClientInfoset method allows you to obtain the infoset for a particular BrokerClient. The setClientInfoset method allows you to set the infoset for a particular BrokerClient. For convenience, the client infoset is treated by both of these methods as a BrokerEvent.

Creating and Destroying Broker Clients

Creating a BrokerClient will establish a connection between your application and a Broker. This connection is identified by the following tuple:

- The name of the host where the Broker is executing.
- The IP port number assigned to the Broker.
- The name of the Broker.

Multiple Broker clients within a single application that connect to the same Broker will all share a single network connection, as described in "Sharing Connections" on page 40.

Your application can create a Broker client by calling the BrokerClient constructor and specifying these parameters:

- The name of the host where the Broker to which you want to connect is executing.
- The name of the Broker to which you want to connect. You specify a `null` value if you want to connect to the *default Broker*. The default Broker for a particular host is determined by your webMethods Broker administrator.

- A unique client ID that identifies your Broker client. You can specify a `null` value if you want the Broker to generate a unique client ID for you. The client ID generated by the Broker can be obtained after this call completes by calling the BrokerClient.getClientId method.
- The client group for your new Broker client. Client groups define the event types your new Broker client will be able to publish or retrieve, as well as the life cycle and queue storage type for your Broker client. Client groups are defined by your webMethods Broker administrator.
- The name of the application that is creating the Broker client. This name is used primarily by webMethods Broker administration and analysis tools. The application name can be any meaningful string of characters you choose.
- A BrokerConnectionDescriptor to be used for the new Broker client. If you specify a `null` value, a new connection descriptor will be created for you. The BrokerConnectionDescriptor class is covered in detail in " Broker Connection Descriptors" on page 35.

The following example is from a sample application that shows the creation of a new BrokerClient object. In this example, a `null`Broker name is passed to indicate that the caller wants to connect to the default Broker on the host "localhost."

Note: See "Parameter Naming Rules" on page 199 for restrictions on Broker names.

A `null` client ID is specified, indicating that the Broker should generate a unique client ID. A client group named "default" is requested and the application name is set to "Publish Sample #1." A `null` connection descriptor is passed, indicating that the caller wants a default connection to be established.

```
import COM.activesw.api.client.*;
class publish1
{ static String broker_host = "localhost";
  static String broker_name = null;
  static String client_group = "default";
  . . .

public static void main(String args[])
{
 BrokerClient c;
 . . .
 /* Create a client */
 try {
 c = new BrokerClient(broker_host, broker_name, null,
 client_group, "Publish Sample #1",null);
 } catch (BrokerException ex) {
 system.out.println("Error on create client\n"+ex);
 return;
}
 . . .
```

Client Identifiers

The client identifier is a string that uniquely identifies your Broker client. The client identifier is used to:

- Identify the Broker client that published an event, described in "Read-only Envelope Fields" on page 72.
- Identify the recipient of a delivered event, described in "Delivering Events" on page 121.
- Re-connect a previously disconnected BrokerClient, described on "Disconnecting and Reconnecting" on page 29.
- Allow two or more Broker clients connected to the same Broker to share the same client state, described on "Sharing Client State" on page 40.

> **Note:** A client identifier cannot start with a '# 'character, nor can it contain '/' or '@' characters. See "Parameter Naming Rules" on page 199 for complete details.

Assigning Client Identifiers

It is always best to let the Broker assign a unique client identifier at the time you create your Broker client. You then retrieve your client identifier by calling the BrokerClient.getClientId method.

Obtaining Client Identifiers

The best way to obtain another Broker client's identifier is by retrieving the `pubId` envelope field from an event published by that client, as described in "Obtaining the Client Identifier" on page 122.

Hard-coding Client Identifiers

In some cases, you have a compelling reason to hard-code a client identifier into your application or to accept the client identifier as a command-line argument:

1. If your Broker client is designed to be disconnected and reconnected over multiple sessions and needs to preserve the client identifier, you might want to hard-code the client identifier.
2. If your application cooperates with other applications that need to know your client identifier, you can choose to hard-code the client identifier into each of the cooperating applications. If the applications are connected to different Brokers in a multi-Broker environment, you must hard-code the fully-qualified client identifier by adding the name of the Broker to which the client is connected to the client identifier, as shown in the table below.

Unqualified and fully qualified client identifiers

Client	Unqualified Client ID	Client's Broker Name	Fully Qualified Client ID
Joe	1000	Denver	//Denver/1000
Mary	1001	Denver	//Denver/1001
Chuck	1000	Chicago	//Chicago/1000

Destroying a Broker Client

The following example shows the use of the destroy method. When a BrokerClient is destroyed, its event queue and all other client state information will also be destroyed.

```
. . .
BrokerClient c;
. . .
try {
c.destroy();
} catch (BrokerException ex) {
system.out.println("Error on client destroy\n"+ex);
return;
}
. . .
```

Using Several Broker Clients

You might find that a number of programming situations are made easier by creating several Broker clients within a single application.

- If your application has several phases of operation, you can use a separate client to represent each phase.
- Because different Broker clients can belong to distinct client groups, they can subscribe to and publish different event types. You can use this feature to divide the work your application needs to perform.
- A multi-threaded application can spawn separate threads for each Broker client, which can result in a cleaner programming model.

Disconnecting and Reconnecting

The webMethods Broker API allows you to disconnect a Broker client from a Broker without actually destroying the Broker client. This is only useful if your Broker client's life cycle is explicit-destroy because the client state for the Broker client and all queued events is preserved by the Broker. Any events received by the Broker for your Broker client while it was disconnected will remain in the Broker client's event queue. When

your Broker client reconnects to the Broker, specifying the same client ID, it can continue retrieving events from the event queue. Disconnecting and reconnecting can be particularly useful for applications that want to do batch-style processing at scheduled intervals.

Disconnecting a Broker Client

You disconnect your Broker client by calling the BrokerClient.disconnect method, as shown in the following example. If the Broker client's life cycle is destroy-on-disconnect, the client state and event queue storage will be destroyed by the Broker. If the Broker client's life cycle is explicit-destroy, the client state and event queue storage will be preserved by the Broker until the Broker client reconnects.

```
. . .
BrokerClient c;
. . .
try {
c.disconnect();
} catch (BrokerException ex) {
System.out.println("Error on client disconnect\n"+ex);
return;
}
. . .
```

Reconnecting a Broker Client

You can reconnect a previously disconnected Broker client that has a life cycle of explicit-destroy by calling the BrokerClient.reconnect method and specifying the same client ID that was used when your client was last connected, as shown in the following example. If the Broker client's life cycle is explicit-destroy, the client state and event queue storage will have been preserved by the Broker since previous call to BrokerClient.disconnect was made.

Note: "Automatic Re-connection" on page 37 describes how you can enable automatic re-connection for your Broker client in the event the connection to the Broker is lost.

```
static String broker_host = "localhost";
 static String broker_name = null;
 static String client_group = "default";
 . . .
 public static void main(String args[])
 {
   BrokerClient c;
   String client_id;
   . . .
   /* Create a client */
   try {
     c = new BrokerClient(broker_host, broker_name, null,
     client_group, "Publish Sample #1",null);
   } catch (BrokerException ex) {
   . . .
   }
   /* Save the client ID */
   client_id = c.getClientId();
```

```
. . .
/* Disconnect the client */
try {
  c.disconnect();
} catch (BrokerException ex) {
. . .
}
. . .
/* Reconnect the client with the same client_id */
try {
  c = BrokerClient.reconnect(broker_host, broker_name,
  client_id, null);
} catch (BrokerException ex) {
. . .
}
. . .
```

Using the newOrReconnect Method

You might find it convenient to use the BrokerClient.newOrReconnect method to create or reconnect a client when your client application is expected to be executed repeatedly. The newOrReconnect method shown in the following example, attempts to create a new Broker client. If the client already exists and was simply disconnected, it will be reconnected.

```
static String broker_host = "localhost";
static String broker_name = null;
static String client_group = "default";
public static void main(String args[])
{
 BrokerClient c;
  String client_id;
  . . .
  client_id = "123";
  /* The first time this program is executed, a Broker client
   * will be created. Subsequent executions will simply reconnect
   * the Broker client.
   */
 try {
    c = BrokerClient.newOrReconnect(broker_host, broker_name, null,
    client_group, "Publish Sample #1",null);
    } catch (BrokerException ex) {
    . . .
    }
 /* Do some processing */
 . . .
 /* Disconnect the client */
 try {
 c.disconnect();
 } catch (BrokerException ex) {
 . . .
 }
 . . .
```

Connection Notification

The connection notification feature allows you to register a callback method for a particular Broker client that will be invoked if the client is disconnected from the Broker.

The connection notification feature can be particularly useful if your Broker client is using the automatic reconnect feature and needs to know when it has been reconnected to the Broker.

Note: Connection callback methods do not have a global scope. They must be registered separately for each Broker client that wants to use this feature.

Defining a Callback Object

You use the BrokerConnectionCallbackinterface to derive your own callback object. Your implementation must provide an implementation of the handleConnectionChange callback method. You can implement this method to recreate any needed client state that have been lost or to provide other methods, such as logging of error messages.

Registering the Callback Object

You use the BrokerClient.registerConnectionCallback method to register a method you want to be called in the event your Broker client is disconnected or reconnected to its Broker. This method accepts two parameters. The first parameter is the BrokerConnectionCallback-derived object that implements your callback method. The second parameter is a client_data object, which is used to pass any needed data to the callback method.

Note: Any callback objects previously registered for a Broker client will be replaced by the one currently being registered.

When the BrokerConnectionCallback object's the callback method is invoked, that method's `connect_state` parameter will be set to one of the BrokerClient-defined values shown in the following table.

Table 1. Connect state values

connect_state	Meaning
`CONNECT_STATE_DISCONNECTED`	The client has been disconnected.
`CONNECT_STATE_CONNECTED`	The Client connection has been re-established, because automatic reconnect was enabled.
`CONNECT_STATE_RECONNECTED`	The Client was disconnected, but the connection was re-established immediately. This only happens if the automatic reconnect feature is enabled and the connection is re-established before a disconnected state can be reported.

Un-Registering Callback Objects

You can un-register a callback by invoking the BrokerClient.registerConnectionCallback method with a null callback object.

Obtaining Client State and Status

There are a variety of methods you can use to obtain state and status information about a Broker client.

Basic Properties

Use the BrokerClient.getBrokerVersionNumber method to obtain the Broker's version number.

Use the BrokerClient.doesSubscriptionExist method to obtain the name of the application associated with a Broker client.

Use the BrokerClient.getBrokerHost method to obtain the name of the host where the Broker that is associated with a Broker client is executing.

Use the BrokerClient.getBrokerName method to obtain the name of the Broker to which a Broker client is connected.

Use the BrokerClient.getBrokerPort method to obtain the IP port number of the Broker to which a Broker client is connected.

Use the BrokerClient.getClientGroup method to obtain the name of the client group with which a Broker client is associated.

Use the BrokerClient.getClientId method to obtain the a Broker client's client ID.

Use the BrokerClient.toString method to obtain a string that contains names of the client, client group, and Broker for a Broker client.

Use the BrokerClient.isConnected method to determine whether a Broker client is currently connected to a Broker.

Use the BrokerClient.isClientPending method to determine whether there are any pending events for a particular Broker client.

Use the BrokerClusterPublisher.getClusterPublisherInfo method to determine whether there are any pending events for any Broker client in use by the application.

Client Queue Methods

Use the BrokerClient.getQueueLength method to determine the number of events currently waiting in a Broker client's event queue.

Use the BrokerClient.clearQueue method to empty the event queue associated with a Broker client.

Important: Use the BrokerClient.clearQueue method with care because it will delete events which have not yet been processed by the Broker client. If multiple clients are sharing the same client state, invoking this method can have far-reaching effects.

Advanced Features

The webMethods Broker Java API provides several advanced methods that you can use to manipulate network connections and to obtain platform information.

Connection Settings

Use the BrokerClient.getDefaultBrokerPort method to obtain the default IP port number that will be used by the BrokerClient constructor and the BrokerClient.reconnect method when connecting to a Broker.

The default port number is used for non-SSL connections and has a value of 6849. The default port number is also used to calculate the port numbers shown in the table below.

Table 2. Port numbers calculated from the default port number

Port number	Description
default port number minus 1	Used for SSL connections without client authentication.
default port number minus 2	Used for SSL connections with client authentication.

Client Time-out Settings

Use the BrokerClient.setDefaultClientTimeout method to set the default time-out for Broker requests. This method will also return the previous time-out settings.

The default connection time-out value is 30 seconds. You might find it useful to lower the connection time-out value in high-performance environments where a delay of 10 seconds probably indicates some sort of failure.

Platform Information

The platform information methods provide a way for your Broker clients to determine the version of the webMethods Broker Java API which they are using as well as the operating system and hardware platform on which they are executing.

Platform information is stored in the form of key-value pairs. The following keys are registered by the webMethods Broker Java API and cannot be changed. You can, however, add your own platform keys.

Key	Value
AdapterLang	"Java"
AdapterLangVersion	<current API version>
OS	The name of the operating system under which the caller is executing.
Hardware	The name of the hardware platform on which the caller is executing.

Use the BrokerClient.getPlatformInfo method to obtain the value associated with a particular key.

Use the BrokerClient.getPlatformInfoKeys method to obtain an array of all the currently defined platform keys.

Use the BrokerClient.setPlatformInfo method to set the value associated with a particular key. If the platform key you specify does not exist, it will be added along with its value.

Broker Territory and Broker-to-Broker Communication

webMethods Broker allows two or more Brokers to share information about their event type definitions and client groups. This enables communication between Broker clients connected to different Brokers. To share information, Brokers join a *territory*. All Brokers within the same territory have knowledge of one another's event type definitions and client groups. For more information on this feature, see *Administering webMethods Broker*.

Use the BrokerClient.getTerritoryName method to obtain the territory name of the Broker to which the Broker client is connected.

Broker Connection Descriptors

The attributes of the connection between a Broker client and a Broker are defined by the BrokerConnectionDescriptor object. You use the attributes in the connection descriptor to:

- Enable or disable the automatic re-connection of a Broker client in the case where the connection to the Broker is lost.
- Enable or disable the sharing of a network connection by more than one Broker client. Only Broker clients located within the same process share a Broker connection.
- Enable or disable the sharing of client state by multiple Broker clients.
- Control the use of the secure socket layer (SSL) for authentication and encryption.
- Enable or disable the redelivery counting feature and indicate whether the redelivery counter will operate in manual or automatic mode.
- Enable or disable the keep-alive feature, which will prevent the client connection from being dropped without the Broker's knowledge.

Note: If you use a BrokerConnectionDescriptor to define the attributes of a connection, you must create the descriptor and set its attributes before you use it to create or reconnect a Broker client. Changing the attributes of a connection descriptor will not affect the Broker client currently using the descriptor, but it will affect any subsequent uses of the descriptor.

Creating Connection Descriptors

You can create a connection descriptor by calling a BrokerConnectionDescriptor constructors. By default, a newly created connection descriptor will have the following attributes:

- Connection sharing is enabled.
- Client state sharing is disabled.
- Event ordering is set to `AW_SHARED_ORDER_BY_PUBLISHER`. See "By-Publisher Event Ordering" on page 41 for a description of event ordering.
- The SSL certificate file is set to `null`.
- The redelivery counting feature is disabled.
- The keep-alive feature is disabled.

Obtaining a Client's Connection Descriptor

The BrokerClient.getConnectionDescriptor method allows you to obtain the connection descriptor for a particular Broker client.

Converting to String

Use the BrokerConnectionDescriptor.toString method to obtain a string containing the information associated with a connection descriptor.

Automatic Re-connection

You can use a connection description to enable automatic re-connection of a Broker client to a Broker in the event that the connection is lost. The automatic re-connection feature is disabled by default.

If your Broker client makes a request to the Broker and the connection has been lost, an attempt is made to reconnect to the Broker, just as if you had invoked the BrokerClient.newOrReconnect method.

You can use the BrokerConnectionDescriptor.setAutomaticReconnect method to enable or disable the automatic re-connection feature.

Use the BrokerConnectionDescriptor.getAutomaticReconnect method to determine whether this feature is currently enabled or disabled.

Using Automatic Re-connection

You can use automatic re-connection, along with the explicit-destroy life cycle and guaranteed storage options, to provide a high degree of reliability in your application design.

Broker clients that use the automatic re-connection feature can put their BrokerClient method invocations inside a retry loop and rely on the connection being established as needed. You might want to insert a time delay into these loops to avoid consuming too much CPU time.

Rules for Automatic Re-connection

Here are some important rules to keep in mind when using the automatic re-connection feature.

- If the Broker disconnects from your client while the client is awaiting a reply from the Broker, a BrokerConnectionClosedException is reflected to the client and no re-connection is attempted.
- If your Broker client is using the callback model for event processing, automatic re-connection will be attempted (when necessary) at any of these points:
  - Each time BrokerClient.dispatch is invoked.
  - Each time BrokerClient.mainLoop is invoked.
  - Each time an event is received by the BrokerClient.threadedCallbacks method on a client thread that is still connected.
- Broker clients with an explicit-destroy life cycle that are automatically reconnected will find their previous client state has been preserved, including previously registered subscriptions, events in the event queue, and all other client state.

  Note: For Broker clients with an explicit-destroy life cycle, previously retrieved events which were not acknowledged will be presented again after

automatic re-connection. When a Broker client that is sharing its state with other clients is disconnected, the other clients retrieve the unacknowledged events. You can avoid receiving duplicate events by explicitly acknowledging events using the BrokerClient.acknowledge, BrokerClient.acknowledgeThrough, or BrokerClient.getEvent methods.

- Broker clients with a destroy-on-disconnect life cycle that are automatically reconnected will find that any previously registered subscriptions, events in the event queue, and all other client state will no longer exist. These Broker clients use the connection notification feature to aid them in recreating their previous subscriptions and client state. For more information, see "Connection Notification" on page 31.

Using Client Keep-Alive

You can use the Broker client keep-alive feature for two purposes:

- To quickly detect dropped client connections. This feature is useful in network configurations that might not notify the Broker until long after a client becomes disconnected (which, with some types of connections might be hours later).
- To keep the client connection open through firewalls.

If client state is not shared, an undetected broken connection does not pose a problem. The Broker will automatically redeliver unacknowledged events to the client when it reconnects. However, when a client's state is shared, the Broker cannot distinguish the reconnection of a disconnected client from the ordinary reconnections of the other clients that are sharing the same state (shared clients all use the same client ID). As a result, the Broker will continue to hold that client's unacknowledged events in the queue until it receives an explicit disconnect notice from the network (generally, when the TCP/IP connection finally times out). If the shared-state client requires ordered events by publisher, the unacknowledged events will also prevent further processing of any additional events from their publishers.

You can avoid this condition by enabling the keep-alive feature in the connection descriptor. When enabled, the keep-alive feature causes the Broker to periodically check its connection to clients that are connected through the descriptor. To test the connection, the Broker sends a keep-alive message to any client that is idle for a specified period of time. If the client does not respond to the keep-alive message within a certain interval, the Broker explicitly disconnects it. By explicitly disconnecting the client, the client's unacknowledged events immediately become available for redelivery and can be retrieved by the other shared-state clients.

Setting the Keep-Alive Parameters

To specify the behavior of the Broker's keep-alive feature, use the BrokerConnectionDescriptor.setKeepAlive method. This method sets the following parameters in the BrokerConnectionDescriptor object:

- **KeepAlivePeriod.** The number of seconds of idle time the Broker waits before issuing a keep-alive message to a client. To suppress keep-alive messages entirely, set *KeepAlivePeriod* to Integer.MAX_VALUE.
- **MaxResponseTime.** The number of seconds the Broker waits for a response from the client after issuing a keep-alive message. This value must be greater than 0 and less that the Keep-Alive Period. Set the Max Keep-Alive Response Time to Integer.MAX_VALUE to specify an infinite response time.
- **RetryCount.** The number of times the Broker resends a keep-alive message to a non-responsive client before disconnecting that client.

How you configure the keep-alive settings depends upon the reason for which you are using keep-alive.

- For zombie session removal, you set a fairly short keep-alive interval, typically in seconds, so that you can quickly remove any zombie sessions.
- For keeping the connection open through firewalls, you set a keep-alive interval closely matching the firewall time-out period, which is typically in minutes. This prevents the firewall from shutting down the connection because of inactivity.

The following code fragment uses the setKeepAlive method to specify a *KeepAlivePeriod* of 90 seconds, a *MaxResponseTime* of 30 seconds, and a *RetryCount* of 5.

```
BrokerConnectionDescriptor desc;

/* Create descriptor */
desc = new BrokerConnectionDescriptor();
desc.setKeepAlive(90,30,5);
```

Be aware that different combinations of the *KeepAlivePeriod* , *MaxResponseTime* , and *RetryCount* produce distinctly different behaviors. For details, see the parameter descriptions for the BrokerConnectionDescriptor.setKeepAlive method.

Important: Client keep-alive uses server resources. If you have a large number of clients, the Broker Server may become bogged down servicing these requests. In situations such as these, increasing the default keep-alive period may be necessary. Make sure to set the keep-alive values appropriately and do not over-use the feature.

Getting the Current Keep-Alive Parameters

Use the following methods to obtain the current values of the keep-alive settings:

- BrokerConnectionDescriptor.getKeepAlivePeriod method.
- BrokerConnectionDescriptor.getKeepAliveResponseTime method.
- BrokerConnectionDescriptor.getKeepAliveRetryCount method.

The following example checks whether the values of *KeepAlivePeriod* and *MaxResponseTime* are set to Integer.MAX_VALUE. (When both *KeepAlivePeriod* and

MaxResponseTime are set to Integer.MAX_VALUE, it disables the keep-alive feature entirely).

```
BrokerConnectionDescriptor desc;
  .
  .
  .
boolean isDisabled;
isDisabled = (desc.getKeepAlivePeriod() == Integer.MAX_VALUE) &&
             (desc.getMaxResponseTime() == Integer.MAX_VALUE) ;
  .
  .
  .
```

Using the Keep-Alive Feature with a Shared Connection

If you want your clients to use the keep-alive feature and also share a connection, all clients sharing the connection must use the same BrokerConnectionDescriptor object or, if using individual BrokerClientDescription objects, must use identical keep-alive parameter values.

Sharing Connections

The webMethods Broker API improves client application performance by having all of your Broker clients that are accessing the same Broker share a single connection. If your client application has special requirements, you override this behavior.

Use the BrokerConnectionDescriptor.setConnectionShare method to enable or disable the sharing of a particular connection by more than one Broker client.

Use the BrokerConnectionDescriptor.getConnectionShare method to determine whether a particular connection be shared by more than one Broker client.

Sharing Client State

webMethods Broker allows Broker clients in different applications to share the same client state. Sharing client state allows several Broker clients, possibly executing on different hosts, to handle events in a parallel, first-come, first-served basis. This provides both parallel processing and load balancing for event handling.

One use for state sharing is load balancing event flows. If you have an application that processes request events (the webMethods Broker dbAdapter is one example), you might want to have several instances of the application available to process these requests.

Broker clients sharing the same client state are treated as one Broker client with regard to the state they are sharing. Any changes to the event subscriptions or event queue, such as clearing the queue, will affect all of the clients sharing the state.

Event Processing with Shared Client State

You have two options for how the Broker will present events to Broker clients that are sharing the same client state. The default event ordering is called *by-publisher* and is described on "By-Publisher Event Ordering" on page 41. You can also specify no ordering, as described on "Event Processing without Ordering" on page 42.

The BrokerConnectionDescriptor.getSharedEventOrdering method returns the current event processing order as either BrokerConnectionDescriptor.`SHARED_ORDER_NONE` or BrokerConnectionDescriptor.`SHARED_ORDER_BY_PUBLISHER`.

Use the BrokerConnectionDescriptor.setSharedEventOrdering method to specify the event processing order that you want.

By-Publisher Event Ordering

The Broker guarantees that events from a single publishing client cannot be processed out of order. This has important implications when several Broker clients are sharing the same event queue. The table below shows a client event queue containing events received from three different publishing clients; Broker client A, Broker client B, and Broker client C.

Publishing Client	Event Queue Position
BrokerClient A	1
BrokerClient B	2
BrokerClient A	3
BrokerClient C	4
BrokerClient B	5
BrokerClient C	6

Consider these steps:

1. BrokerClient X receives the event from queue position 1 without acknowledging the event.
2. BrokerClient Y receives the event from queue position 2 without acknowledging the event.
3. When BrokerClient Y then asks for another event, it is given the event from queue position 4 because the last event published by Client A has not yet been

acknowledged. For more information on acknowledging events, see "Using Sequence Numbers" on page 133.

Event Processing without Ordering

Invoke the BrokerConnectionDescriptor.getSharedEventOrdering method, specifying a value of `SHARED_ORDER_NONE`, to indicate that you do not want the Broker to guarantee the order of event processing. With an event ordering of `SHARED_ORDER_NONE`, events will be presented to any of the Broker clients sharing the client queue in the order in which they appear in the queue.

When is Sharing State Useful?

A shared-state client is useful only if it receives events that it can process in parallel. For example, a shared-state client makes sense if it will receive events from one or more publishers and it can process those events in any order (i.e., SHARED_ORDER_NONE). It also makes sense in cases where events must be processed in order by publisher (i.e., SHARED_ORDER_BY_PUBLISHER), and the events come from multiple publishers.

A share-state client is not useful in a situation where it receives events from only a single publisher and those events must be processed in order (i.e., SHARED_ORDER_BY_PUBLISHER). Under these circumstances, the shared-state client would be required to process all events serially. It would provide no performance benefits, because, in this case, the shared-state client would never receive any events that it could process in parallel.

State Sharing Methods

Use the BrokerConnectionDescriptor.getStateShare method to determine whether client state sharing is enabled or disabled for a particular connection descriptor.

Use the BrokerConnectionDescriptor.setStateShare method to enable or disable the sharing of the client state associated with the connection descriptor.

In addition to these methods, you can use the BrokerClient.getConnectionShareLimit and BrokerClient.setConnectionShareLimit methods to obtain and limit the number of Broker clients that can share a particular client state.

Calling Sequence for Sharing Client State

You can follow the steps below to allow multiple Broker clients to share the same client state.

1. Create a new BrokerConnectionDescriptor and set the state sharing to `true`.
2. Create a new BrokerClient, passing a client ID and the descriptor created in step 1.
3. After the Broker client is created with shared state, other Broker clients using the same Broker can use the BrokerClient.reconnect method, along with the client ID from step 2, to connect to the Broker client. No special BrokerConnectionDescriptor settings are required when reconnecting.

> **Note:** All Broker clients that are to share the same state must be connected to the same Broker.

Priority Ordering for Broker Queues

The BrokerEvent.getPriority method and BrokerEvent.setPriority method allows you to order the documents in a queue by priority. To use priority ordering:

- A publisher assigns the priority to the messages that it publishes and
- The client enables the priority queue.

> **Note:** After you create a client with priority queue enabled, it is always priority enabled. You *cannot* turn the priority enabling off.

For Broker, implementation of priority messaging is based on server-side implementation of the priority queue. Each document is queued for delivery to the subscriber in the following order: (1) priority and (2) publication time. Documents are ordered by priority at the destination queue when the document is inserted into the destination client's queue, if that queue has priority ordering enabled. Priority ordering does not apply to forward queues, when documents are forwarded to subscriptions in territories, so there is a possibility that lower priority documents are delivered before incoming higher priority documents are delivered to the queue.

For priority values, Broker uses the same values according to JMS standards: priority values are from 0-9, where 0 is the lowest priority and 9 is the highest priority (expedited processing). The default value is 4.

Broker maintains the priorities through a set of priority cursors. These priority cursors are *not* persisted to disk, so the cursors are lost when you restart Broker. As a result, when you shut down Broker and restart it:

- All the priority cursors are reset to empty.
- All the existing documents in the queue are delivered with the default priority of 4.
- Any arriving documents are prioritized and delivered in priority order.

This means that after restarting the Broker, the priority of any new document places that document relative to the priority of the old documents, which now all have a default value of 4. For example, if you have a document A with a priority of 9 and you have to restart the Broker, document A and all of the documents still in the queue are now assigned the default priority of 4. When the queue receives any new documents after restarting the Broker, priority ordering applies to those new documents. So, if after restart, the new document B has a priority of 7, document B will now have a higher priority than document A, even though document A initially had a higher priority of 9.

Priority Ordering and Performance

All client queues have a retrieval cursor. This cursor speeds up access to messages when there are a large number of unacknowleged messages in the queue that are not candidates for retrieval.

In situations where there are a large number of unacknowledged messages in the queue (such as might happen with JMS clients that control client acknowledgment), it is possible for messages to be inserted into the client queue ahead of the retrieval cursor, regardless of their priority.

To remedy this situation, Broker resets the retrieval cursor back to the beginning of the client queue to re-evaluate all messages in the queue after an insert operation is made. If the number of messages in the queue is small, the impact of the reset operation on performance is negligible; however, if the number of messages is large, there could be noticeable effect on performance. Therefore, if your client queues maintain a large number of unacknowledged messages, consider turning priority messaging off if you notice an unacceptable impact on performance.

Using Security Features

For more information on enabling basic authentication, see"Working with Basic Authentication" on page 181.

For more information on enabling SSL features, see "Working with SSL" on page 185.

Redelivery Counting

The BrokerConnectionDescriptor.setAutomaticRedeliveryCount method and the BrokerConnectionDescriptor.setRedeliveryCountEnabled method enables the redelivery counting feature for a BrokerClient. When redelivery counting behavior is enabled, the Broker maintains a counter for each guaranteed event that it sends to the BrokerClient. By testing the value in the counter, your client can determine whether it has received an instance of a particular event before.

The redelivery feature can be used in *manual* or *automatic* mode. The mode you select determines whether the redelivery counter is incremented by the Broker or by your client. When you use automatic mode, the Broker updates the redelivery counter prior to sending a event to the client. When you enable manual mode, your client must explicitly increment the redelivery counter when it receives an event.

To check whether redelivery counting is enabled, you use the BrokerConnectionDescriptor.getAutomaticRedeliveryCount method and the BrokerConnectionDescriptor.getRedeliveryCountEnabled method.

Note: A redelivery counter is maintained for guaranteed events only. Volatile events always has a redelivery count of `-1`. Additionally, if a client does not enable

redelivery counting, all events that it receives will have a redelivery count of `-1`.

For more information about using the redelivery counting feature, see "Detecting Redelivered Events" on page 85.

3 Using the Callback Model

Overview

This chapter describes the webMethods Broker Java API for receiving and processing events using the callback model. Reading this chapter will help you to understand:

- How to register general and specific callback objects.
- How to derive your own callback object from the BrokerCallback interface.
- How to retrieve and process events from the Broker using callback objects.

Understanding Callbacks

The callback model for processing event types allows your client application to register one or more callback objects to process event types for a Broker client. Unlike the get-events model, which can only process event types for one Broker client at a time, the callback model can receive any event type for any of your client application's Broker clients and then dispatch it to the appropriate callback object's event handling method. If your client application creates several Broker clients, using the callback model frees your application from making separate calls to one of the webMethods Broker get-event methods for each Broker client.

Using Callbacks

Follow these steps to use the callback processing model:

1. Declare your callback objects
2. Verify that the subscription(s) that you want is (are) allowed using BrokerClient.canSubscribe.
3. Register the subscription(s) using one of the BrokerClient.newSubscription methods.
4. Register a *general* callback object using BrokerClient.registerCallback.
5. Next, you can optionally register any *specific* callback objects you want, using:
   - BrokerClient.registerCallbackForSubId
   - BrokerClient.registerCallbackForTag
6. Process events using one of the following methods:
   - BrokerClient.dispatch
   - BrokerClient.mainLoop
   - BrokerClient.threadedCallbacks
7. Cancel all the callbacks using BrokerClient.cancelCallbacks, cancelCallbackForSubId, or cancelCallbackForTag.

8. Cancel all the subscriptions using one of the BrokerClient.cancelSubscription methods.

Implementing a Callback

All callback objects that you register must be derived from the BrokerCallback interface. When deriving your own class from BrokerCallback, you must provide an implementation for the handleBrokerEvent method. This method has the following declaration:

```
public boolean handleBrokerEvent(
  BrokerClient client,
  BrokerEvent event,
  Object client_data);
```

client The client for which the event has been received.

event The event that is being dispatched to this method.

client_data Any data that you want to be passed to this method when it is invoked.

Your method should return `true` if its processing was successful or `false` if a failure occurred. If `true` is returned, the event will be acknowledged automatically. For information on acknowledging events, see "Using Sequence Numbers" on page 133.

Note: If you want to save a copy of an event passed to a callback method, you must use the BrokerEvent copy constructor.

Passing Arguments to Callback Methods

When you register your callback object, you can specify a *client_data* object that might be necessary for the callback method to complete its processing. The *client_data* parameter might refer to state information that the callback method must access and update each time it is invoked.

Assume that you want to count the number of events that your application processes. When you register your callback object, you could use the *client_data* value to refer to the counter variable. When the callback object's event handling method is invoked, it can increment the counter.

The following example illustrates how to set up an argument for a callback object:

```
class IntHolder {
  int count;
};
static IntHolder counter;
. . .
public static void main(String args[])
{
  int n;
  counter.count = 0;
  SampleCallback sample_callback = new SampleCallback();
  . . .
```

```
  /* Check if can subscribe */
  . . .
  /* Register callback */
  try {
    c.registerCallback(sample_callback, counter);
  } catch (BrokerException ex) {
    System.out.println("Error on registering callback\n"+ex);
        return;
    }
  . . .
```

The following example illustrates the SampleCallback implementation:

```
public boolean handleBrokerEvent(BrokerClient c, BrokerEvent e,
    Object counter)
{
  /* increment counter */
  counter.count++;
  /* perform rest of event processing */
  . . .
}
```

General Callback Objects

When using the callback model, you must register a *general* callback object for each Broker client by calling the BrokerClient.registerCallback method. When an event is received, if there are no specific callback objects registered which match the event's subscription identifier or tag, the general callback object's handleBrokerEvent method will be invoked to handle the event.

You can have a single callback object process all of your application's events by making a separate BrokerClient.registerCallback call for each BrokerClient that your application uses, specifying the same callback object each time.

Depending on the complexity of your design, you might find that a single, general callback object is all that your client application requires.

Note: You must register a general callback object before registering any specific callback object.

Using General Callbacks

The following example contains an excerpt from a sample that shows the use of the BrokerClient.registerCallback method to register a general callback object.

```
class subscribe2
{
  . . .
  static int num_to_receive = 10;
  public static void main(String args[])
  {
  . . .
  SampleCallback2 sample_callback =
  new SampleCallback2(num_to_receive);
  /* Create a client */
  . . .
```

```
/* Check if can subscribe */
. . .
/* Register callback */
try {
  c.registerCallback(sample_callback,null);
} catch (BrokerException ex) {
 System.out.println("Error on registering callback\n"+ex);
 return;
 }
/* Open the subscription */
 . . .
/* Do main loop */
  try {
   BrokerClient.mainLoop();
 } catch (BrokerException ex) {
 System.out.println("Error on dispatch\n"+ex);
 return;
}
. . .
```

Specific Callback Objects

Depending on the complexity of your design, you might find that a single, general callback object is all that your client application requires. More complicated designs might need to make use of specific callback objects.

A *specific* callback object is only used to process an event with a particular subscription identifier or event tag for a particular Broker client. You can register a specific callback object by calling either the BrokerClient.registerCallbackForSubId method, or BrokerClient.registerCallbackForTag method.

Event tag fields are part of the request-reply model, described in "Using Request-Reply" on page 97.

Note: If a received event matches the criteria for more than one callback, each callback objects handleBrokerEvent method will be invoked to handle the event.

Using Specific Callbacks

The following example illustrates how to register a specific callback object using the use of the BrokerClient.registerCallbackForSubId method.

```
. . .
public static void main(String args[])
{
. . .
SampleCallback1 general_callback =
new SampleCallback1(num_to_receive);
SampleCallback2 specific_callback =
new SampleCallback2(num_to_receive);
/* Create a client */
. . .
/* Check if can subscribe */
. . .
/* Register general callback */
```

```
  try {
  c.registerCallback(general_callback,null);
  } catch (BrokerException ex) {
    System.out.println("Error on registering general callback\n"+ex);
    return;
  }
  /* Register specific callback with a subscription ID of 1 */
  try {
  c.registerCallbackForSubId(1, specific_callback,null);
  } catch (BrokerException ex) {
  System.out.println("Error on registering specific callback\n"+ex);
  return;
  }
  /* Open the subscription with a subscription ID of 1 */
  try {
  c.newSubscription( 1, "Sample::SimpleEvent",null);
  } catch (BrokerException ex) {
  System.out.println("Error on create subscription #1\n"+ex);
  return;
  }
  /* Do main loop */
  try {
  BrokerClient.mainLoop();
  } catch (BrokerException ex) {
  System.out.println("Error on dispatch\n"+ex);
  return;
  }
  . . .
```

Dispatching Callback Methods

After registering your subscriptions callback object, your client application should then call one of the webMethods Broker dispatching methods to receive events and dispatch the appropriate callback object's event handling method. Only one of the following dispatching methods should be used.

Using dispatch

The BrokerClient.dispatch method can be used to wait to receive a single event for any Broker client, dispatch the event to the appropriate callback object's event handling method, and then return. The example below contains an excerpt from a sample application that shows how to use the BrokerClient.dispatch method.

```
/* Do dispatch loop */
  count = 1;
  while (count <= num_to_receive) {
  try {
   BrokerClient.dispatch(-1);
   } catch (BrokerException ex) {
  System.out.println("Error on dispatch\n" + ex);
  return;
  }
  ++count;
  }
```

Using mainLoop

The BrokerClient.mainLoop method is similar to BrokerClient.dispatch, except that it enters an event loop that will continue to receive and dispatch events until BrokerClient.stopMainLoop is called. The example below contains an excerpt from a sample application that shows how to use the BrokerClient.mainLoop method.

```
/* Do main  loop */
  try {
  BrokerClient.mainLoop();
  } catch (BrokerException ex) {
  System.out.println("Error on dispatch\n" + ex);
  return;
  }
```

Using threadedCallbacks

The BrokerClient.threadedCallbacks method will create a thread that invokes BrokerClient.mainLoop. Using this method can simplify your application code because it handles the thread creation for you. The example below contains an excerpt from a sample application that shows how to use the BrokerClient.mainLoop method.

```
/* Enable threaded callbacks */
  try {
  BrokerClient.threadedCallbacks(true);
  } catch (BrokerException ex) {
  System.out.println("Error on enabling threaded callbacks\n" + ex);
  return;
  }
```

Invoking threadedCallbacks(`true`) is identical to creating a thread and invoking the BrokerClient.mainLoop method on that thread.

Invoking threadedCallbacks(`false`) is identical to invoking the BrokerClient.stopMainLoop method.

Event Dispatching Rules

When an event is received in the callback model, the following rules are used to dispatch the event.

1. If the received event has a tag field and the tag matches a registered callback, the received event is dispatched to that callback object's event handling method.
2. If the received event has a subscription identifier, the event is dispatched once to each callback object that matches the subscription identifier. If the event matches two event subscriptions with the same subscription identifier, the event will be dispatched twice to the callback object for that identifier.
3. The received event will be dispatched to the general callback object's event handling method if the tag did not match any callback and:
   a. At least one subscription identifier did not match a specific callback.

b. No subscription identifiers were matched because the event was delivered.

4 Creating and Initializing Events

Overview

This chapter describes the webMethods Broker objects and methods for creating BrokerEvents and setting their field values. Reading this chapter will help you understand:

- Event types and event names.
- Event fields, including regular fields, sequence fields and structure fields.
- How to create a BrokerEvent object.
- How to set event field values.
- How to obtain the value from an event field.
- Event envelope fields.
- How to set and obtain envelope fields.
- How to quickly populate a BrokerEvent object's fields from a storage object you define.
- How to quickly populate a storage object you define with the field values from a BrokerEvent object.
- How to improve performance by creating BrokerEvent objects that use storage classes.

Event Overview

The first step in implementing your application is to define the content of the events that will be used to communicate information between your client applications and the Broker. Events are defined using Software AG Designer, described in the *Software AG Designer Online Help*.

Note: Broker events are known as Broker*document types* in Designer.

Events contain data that allow applications to communicate with each other. An event can be used to represent:

- Requests to a database application to retrieve or write data.
- Banking transactions such as deposits, withdrawals, and transfers.
- News headlines, stock quotes, or money exchange rates.
- Customer orders, invoices, packing slips, or credit memos.

Event Types

An *event type* defines the event's name as well as the fields that it contains. webMethods Broker API objects, like BrokerEvent, use event type definitions to verify that an event's

fields are set with values that match their defined types. Event type definitions are also used if you call the BrokerEvent.validate method to validate an event's fields.

Note: Events created without a BrokerClient context are not type checked. See "Field Type Checking" on page 59 for more information.

Event Type Cache

The webMethods Broker API copies the event type definitions used by your application from the Broker to which you are connected into a local *event type cache*. The cache improves the performance of the event field type checking process and its use is usually transparent to you. For more detailed information, see "Managing Event Types" on page 139.

Event Type Names

Each event type has a unique name that distinguishes it from other event types.

Note: "Parameter Naming Rules" on page 199 describes the restrictions on event type names.

Event types are organized into *event families*, allowing you to group all of the events related to a particular application domain. An event type name consists of two components; a *scope* and a *base name*.

```
Scope::BaseName
```

The scope component can consist of one or more levels. Consider the following fully qualified event type name.

```
WesternRegion::Hardware::Sales::receiveCustOrder
```

The base name for this event type would be `receiveCustOrder`.

The scope would be `WesternRegion::Hardware::Sales`.

Use the BrokerEvent.getTypeName method to obtain the fully-qualified name for an event type.

Use the BrokerEvent.getBaseTypeName method to obtain the name for an event type event type without the scope qualification.

Use the BrokerClient.getEventTypeNames method to obtain the fully-qualified names of all the event types known to the Broker to which your Broker client is connected.

Use the BrokerClient.getScopeNames method to obtain the fully-qualified names of all the event types known to the Broker to which your Broker client is connected. All event scopes that contain at least one event type will be returned.

Note: Only the names of the event types which your client is permitted to browse are returned by the getEventTypeNames and getScopeNames methods. In most

cases, this corresponds to the set of event types which your client can publish or for which it can register subscriptions.

Use the BrokerEvent.getScopeTypeName method to obtain the scope name from a particular event.

Event Fields

Every event contains *envelope* fields and *data* fields. Envelope fields are consistent for all event types and contain details about the event's sender, destination, and its transit. Envelope fields are described on "Envelope Fields" on page 69.

Event data fields contain the data that your client applications use to exchange information. Event data fields can contain a single value, a sequence of values with the same type, or a structure containing values of different types, or a pre-defined Event Type. Event data fields are discussed on "Event Data Fields" on page 60.

Note: "Parameter Naming Rules" on page 199 describes the restrictions on event field names.

Event Identifier

When an event is published, described in "Publishing and Delivering Events" on page 119, the Broker will assign the event an event identifier that you can use to determine whether two events are exactly the same. You can also use the event identifier to match events with trace events or activity traces, described in the *webMethods Broker Administration Java API Programmer's Guide*. An event identifier is almost certainly unique. It is possible, though extremely unlikely, for two Brokers to generate the same event identifier.

You can retrieve the event identifier using the BrokerEvent.getEventId method.

Creating Events

Before your client application can publish or deliver an event, it must create the event and set the event's fields. The example below contains an excerpt from a sample application that shows the creation of a new event. The constructor takes the following parameters:

- A BrokerClient reference.
- The name of the event type. In the following example, the event scope is "Sample" and the event type name is "SimpleEvent":

```
. . .
public static void main(String args[])
{
BrokerClient c;
BrokerEvent e;
```

```
/* Create a client */
. . .
/* Create the event */
try {
e = new BrokerEvent(c, "Sample::SimpleEvent");
} catch (BrokerException ex) {
System.out.println("Error on create event\n"+ex);
return;
}
. . .
```

General Event Methods

After creating an event, you can use the BrokerEvent copy constructor to create a new event, copying the contents of an existing event to the new event.

Use the BrokerEvent.hasBeenModified method to determine whether the contents of an event have been modified.

Use the BrokerEvent.clearModificationFlag method to mark an event as having not been modified.

Use the BrokerEvent.setModificationFlag method to mark an event as having been modified.

Use the BrokerEvent.toString method to format an event's fields into a string.

Use the BrokerEvent.toFormattedString method to format selected event fields into a string. One version of this method allows you to specify the Locale you want to be used in formatting the string.

Use the BrokerEvent.toLocalizedFormattedString method to format selected event fields into a string using the Locale that you specify.

Use the BrokerEvent.toLocalizedString method to format selected event fields into a string using the current Locale.

Use the BrokerEvent.getClient method to obtain the Broker client associated with an event.

Field Type Checking

When you create an event with a BrokerClient reference, the following field type checking rules will be applied to the event.

1. All event fields will appear to be set on the event at the time the event is created.
2. You are not allowed to set a field which does not exist for the event type.
3. You cannot set an event field with a data type other than that defined by the event type.

When you create an event with a `null`BrokerClient reference, the following type checking rules will be applied to the event.

1. You can set fields with any field name and any data type.

2. Any attempt to retrieve an event field that was not previously set will cause a BrokerFieldNotFoundException exception to be thrown.
3. After a field has been set, you are not allowed to change the field's type without first clearing the event field, using the BrokerEvent.clearField method or clearing the entire event using the BrokerEvent.clear method.

> **Note:** For most applications, you should create an event within the context of a Broker client so that type checking will occur.

Converting Events as Binary Data

Use the BrokerEvent.toBinData method to obtain a binary array representation of a BrokerEvent that is suitable for saving to disk.

Restoring Events from Binary Data

Use the BrokerEvent.fromBinData method to initialize a BrokerEvent from a binary array that was generated from an earlier call to BrokerEvent.toBinData. When re-creating the event, you can specify an associated BrokerClient if you want the event's contents to be type checked.

Event Data Fields

Data fields contain application-specific data that your application will set before publishing an event or will retrieve when processing a received event. Each field has a name and a specifically typed value. Event data field values are stored in a platform-independent representation that allows client applications executing on different hardware platforms and under different operating systems to easily exchange data without worrying about bit significance or byte ordering. The conversion from webMethods Broker's platform-independent data representation to the local data representation is handled transparently by the BrokerEvent get and set methods.

Field Data Types

The following table shows the data types used by client applications for regular, sequence struct, and event data fields.

Event Type Editor Type	Java Language Type	Description
`boolean`	`boolean`	1 (true) or 0 (false), stored as a single byte.

Event Type Editor Type	Java Language Type	Description
byte	byte	Signed 8-bit integer.
char	char	A single character.
date	BrokerDate	An object representing the year, month, day, hour, minute, second, and millisecond.
double	double	Double-precision floating point number.
event		A separate, previously defined, event type.
float	float	Standard-precision floating point number.
int	int	Signed 32-bit integer.
long	long	Signed 64-bit integer.
sequence		Sequence of any regular data types (such as, boolean, byte, char, date, etc.)
short	short	Signed 16-bit integer.
string	String	A string of characters.
struct		A set of basic Java types
unicode char	char	Used for unicode, char (double byte)
unicode string	String	Used for unicode, double byte
unknown		If the application is unable to identify a field, then it is categorized at "unknown."

Use the BrokerTypeDef.getFieldType method to obtain an event field's type.

Obtaining Field Names

You can use the BrokerEvent.getFieldNames method to obtain the names of all the fields contained in a particular event.

Regular Data Fields

A regular data field in an event contains a single value that is obtained and set using the field name and an appropriately typed source or target value.

Setting Regular Data Fields

Several methods are provided for setting a regular data field, based on it's type. These methods are described in BrokerEvent.set<type>Field. The source value for the field being set depends on the field's type, as shown in the table below.

BrokerEvent Method	Value Type
`setBooleanField`	`boolean`
`setByteField`	`byte`
`setCharField`	`char`
`setDateField`	`BrokerDate`
`setDoubleField`	`double`
`setFloatField`	`float`
`setIntegerField`	`int`
`setLongField`	`long`
`setShortField`	`short`
`setStringField`	`String`

The following example contains an excerpt from a sample application that shows the use of the BrokerEvent.setIntegerField method. This method takes the following parameters:

- The name of the event field to be set.

- The value for the field.

```
BrokerEvent e;
int count;
. . .
count = 1;
/* Publish */
while (count < num_to_publish) {
try {
e.setIntegerField("count",count);
} catch (BrokerException ex) {
System.out.println("Error on setting event field\n"+ex);
return;
}
. . .
```

If you attempt to set a field with a value that does not match its defined type, an exception might be thrown. Type checking will not occur if the event whose field is being set was created without a Broker client context, as described in "Field Type Checking" on page 59.

You can also use the BrokerEvent.setField method to set an event field.

Getting Regular Field Values

Several methods are provided for obtaining the value of a regular event data field, based on its type. These methods are described in BrokerEvent.get<type>Field. The type of the value being retrieved depends on the field's type, as shown in the table below.

BrokerEvent Method	Value Type
getBooleanField	boolean
getByteField	byte
getCharField	char
getDateField	BrokerDate
getDoubleField	double
getFloatField	float
getIntegerField	int
getLongField	long
getShortField	short

BrokerEvent Method	Value Type
getStringField	String

The following example contains an excerpt from a sample application that shows the use of the BrokerEvent.getIntegerField method. This method requires the name of the event field you want to retrieve.

```
BrokerEvent e;
int pub_count;
. . .
try {
  pub_count = e.getIntegerField("count");
} catch (BrokerException ex) {
System.out.println("Error on getting count field\n"+ex);
return;
    }
. . .
```

If you attempt to get a field value with a type that does not match the field's defined type, an exception might be thrown. Type checking will not occur if the event whose field is being obtained was created without a Broker client, as described in "Field Type Checking" on page 59.

You can also use the BrokerEvent.getField method to obtain the value and type of an event field.

Sequence Data Fields

Sequence data fields contain a sequence of several values with the same data type. A sequence field can contain any of the types mentioned for simple data fields. A multiple dimension array of values can be represented as a sequence within a sequence.

Setting Sequence Fields

Several methods are provided for setting the values of the entire sequence, or a subset of the sequence. These methods are described in BrokerEvent.set<type>SeqField.

You can also use the BrokerEvent.set<type>Field to set a single value within a sequence field by specifying the field name with an index.

The following example shows the use of the BrokerEvent.setIntegerSeqField method. This method takes the following parameters:

- The name of the event sequence field to be set.
- The number of elements to skip from the beginning of the source array parameter.
- The number of elements to skip from the beginning of the target sequence in the event.
- The number of elements to be set.

- The source array of values of the appropriate type.

  Setting the values of an event sequence field

```
int count[5] = { 0, 1, 2, 3, 4};
   . . .
   try {
   e.setIntegerSeqField("count", 0, 0, 5, count);
   } catch (BrokerException ex) {
 System.out.println("Error on setting sequence field\n"+ex);
 return;
 }
 . . .
```

The manner in which the source array is specified depends on the array's type.

BrokerEvent Method Name	Sequence Value Type
setBooleanSeqField	boolean[]
setByteSeqField	byte[]
setCharSeqField	char[]
setDateSeqField	BrokerDate[]
setDoubleSeqField	double []
setFloatSeqField	float[]
setIntegerSeqField	int[]
setLongSeqField	long[]
setShortSeqField	short[]
setStringSeqField	string[]

Each of the BrokerEvent.set<type>SeqField methods can overwrite all or part of the destination sequence field. After the code shown in the example in "Setting Sequence Fields" on page 64 is executed, the sequence will contain:

```
[0 1 2 3 4]
```

If you then set three elements (3, 2, 1) into this same sequence at location 1, the sequence would then appear as:

```
[0 3 2 1 4]
```

An error will be returned if you attempt to set a field with a value that does not match its defined type.

These methods can also cause the destination sequence to grow in size, if a larger number of elements are stored into the sequence.

These methods *never* reduce the number of elements in the destination sequence. Use the BrokerEvent.setSequenceFieldSize method to reduce the size of a sequence field.

You can also use the BrokerEvent.setSequenceField method to set a sequence field.

Getting Sequence Field Values

Several methods are provided for obtaining all of the values from a sequence event field, or a subset of the sequence, with a single method call. These methods are described in BrokerEvent.get<type>SeqField.

You can also use the BrokerEvent.get<type>Field to obtain a single value from a sequence field by specifying the field name with an index.

The following example shows the use of the BrokerEvent.getIntegerSeqField method. This method takes the following parameters:

- The name of the event sequence field being accessed.
- The number of elements to skip from the beginning of the sequence field in the event.
- The number of source elements to be retrieved.

```
int count_array[];
int num_retrieved;
. . .
try {
count_array = e.getIntegerSeqField("count", 0, 5);
} catch (BrokerException ex) {
System.out.println("Error on getting sequence field\n"+ex);
return;
}
. . .
```

The manner in which the target array is specified depends on the sequence's type.

BrokerEvent Method Name	Field Value Type
getBooleanSeqField	boolean[]
getByteSeqField	byte[]
getCharSeqField	char[]
getDateSeqField	BrokerDate[]

BrokerEvent Method Name	Field Value Type
getDoubleSeqField	double[]
getFloatSeqField	float[]
getIntegerSeqField	int[]
getLongSeqField	long[]
getShortSeqField	short[]
getStringSeqField	String[]

You can also use the BrokerEvent.getSequenceField method to obtain the contents of a sequence field. The BrokerEvent.getSequenceFieldSize method can be used to obtain the number of elements contained in a sequence field.

Structure Data Fields

Structure data fields represent event fields with a user-defined type. A structure data field can contain members that are simple data types, arrays, or other structures.

```
Sample:StructEvent {
string count;
struct sample_struct {
BrokerDate sample_date;
int sample_array [] [];
}
}
```

Setting Structure Fields

Two methods, BrokerEvent.setStructFieldFromEvent and BrokerEvent.setStructSeqFieldFromEvents, are provided for setting all of the values within a structure field. The first method sets the entire contents of a single structure field and the second method sets the entire contents of a *sequence* of structure fields.

You can also use the BrokerEvent.set<type>Field to set the value of a single field within a structure field by specifying the appropriate field name.

Setting a Struct Field from an Event

Because structure fields can contain other fields, the BrokerEvent.setStructFieldFromEvent method allows you to set all of those contained values with just one method call. To use this method, follow these steps.

1. Create an empty event of any event type, using the BrokerEvent constructor. Because the data you will put into this event is not likely to match a real event type definition, create this event without a Broker client so it will not be type checked.
2. Set the fields and values of the event created in step 1 so that they match the fields in the structure you want to set. Use the BrokerEvent.set<type>Field and BrokerEvent.set<type>SeqField methods.
3. Call the BrokerEvent.setStructFieldFromEvent method, passing the event created in the above steps as the source value.
4. Envelope fields on the source event are ignored.

Setting a Struct Sequence Field from an Event

Similar in concept to BrokerEvent.setStructFieldFromEvent, the BrokerEvent.setStructSeqFieldFromEvents method sets a sequence of structure fields from an array of events. Envelope fields on the source events are ignored.

As with the other methods for setting sequence fields, this method might overwrite all or part of the destination structure sequence field. It might also increase the size of the target sequence, but it will never reduce the size of the sequence. To reduce the size of a sequence field, use the BrokerEvent.setSequenceFieldSize method.

Getting Structure Field Values

Two methods, BrokerEvent.getStructFieldAsEvent and BrokerEvent.getStructSeqFieldAsEvents, are provided for obtaining some or all of the values from a structure field in a single method call. The first method obtains all of the values of a single structure field and the second method obtains all the values from a sequence of structure fields.

You can also use the BrokerEvent.get<type>Field to obtain the value of a single field within a structure field by specifying the appropriate field name.

Getting a Struct Field as an Event

Because structure fields can contain other fields, the BrokerEvent.getStructFieldAsEvent method allows you to obtain all of those contained values with just one method call.

As described on "Field Type Checking" on page 59, the retrieved event that represents the structure is not type checked.

Getting a Struct Sequence Field

Similar in concept to BrokerEvent.getStructFieldAsEvent, the BrokerEvent.getStructSeqFieldAsEvents method obtains a sequence of structure fields as an array of events. Envelope fields on the target events are ignored.

As described on "Field Type Checking" on page 59, the retrieved events that represent the structures are not type checked.

Envelope Fields

Many of the event envelope fields are managed for you by the webMethods Broker API methods. Some event fields can be set by your application and others contain values set by the Broker. You cannot set the envelope fields that are set by the Broker, but you can retrieve their values.

Note: Unlike other event fields, attempting to retrieve an envelope field that has not been set will cause a BrokerFieldNotFoundException to be thrown.

Envelope Field	Event Editor Type	Description
`activation`	`unicode_string`	A unique identifier set by the event's publisher to identify a one-time execution of an integration solution.
`appSeqn`	`int`	Sequence number, set by the event's publisher. Your application defines how this field is used. The conventional way to use this field is to count upward from 1.
`appLastSeqn`	`int`	Sequence number, set by the event's publisher. Your application defines how this field is used. Generally used to identify the last event in a sequence of events.
`appPassword`	`unicode_string`	Password of user in appUserName. Used if the resource that processes the event requires authentication.
`appUsername`	`unicode_string`	Represents a user's name.
`businessContext`	`unicode_string`	Used internally to track business process context and audit context.
`controlLabel`	`short[]`	Represents the access label that a receiving client must have to receive the event.
`errorsTo`	`unicode_string`	The client ID to which the event should be forwarded if any errors are

Envelope Field	Event Editor Type	Description
		generated when this event should be sent, instead of sending to the originator of the request.
errorRequestsTo	unicode_string	The client ID to which a request event will be forwarded if any errors generated processing the request. If field is not set, any request event that generates an error will be discarded.
eventTraceInfo	unicode_string	Used internally to append the trace information to documents. This trace information provides insight about the flow of document through the different systems. Used by webMethodsInsight. This field can be used by your application if webMethodsInsight is not tracing the document flow.
locale	unicode_string	Locale of the publishing client expressed as a URN (Uniform Resource Name).
maxResults	int	The maximum number of reply events a requestor would like to receive. A value of 0 indicates that no reply or acknowledgment should be sent for this event.
priority	int	Message priority of values from 0-9, where 0 is the lowest priority and 9 is the highest priority (expedited processing). The default is 4.
replyTo	unicode_string	The client ID to which the replies to this event should be sent, instead of sending to the originator of the request.
signature	byte[]	A byte sequence that holds a digital signature.

Envelope Field	Event Editor Type	Description
`signatureType`	`unicode_string`	Describes the type of digital signature being used.
`startResult`	`int`	A value >= 0 that specifies the starting number of the event to receive. Often used in conjunction with maxResults.
`tag`	`int`	Used in the request-reply model, described in "Using Request-Reply" on page 97, to match a request event with its corresponding reply event.
`trackId`	`unicode_string`	This field's value can be set by a publishing client application to a unique identifier for tracking purposes. This allows an event that is re-published to be tracked. It also allows multiple events associated with a single logical transaction to be tracked. If not set, this field should be treated as if it contained the same value as the `pubId` field.
`transactionId`	`unicode_string`	This field's value can be set by a publishing client application to indicate that an event is part of a transaction. See "Transaction Semantics" on page 109 for more information.
`transformState`	`unicode_string`	This field allows clients that transform data to mark an event's current state. An event could be published with a `transformState` value of "`USEnglish`". A receiving client could translate the event into French and publish it with a `transformState` value of "`French`".

The `appSeqn` and `appLastSeqn` can be used by your publisher and subscriber applications for whatever purpose they require. One possibility is to track a sequence of events which represent a response to a single request event. Your publisher could start `appSeqn` at 1 and set `appLastSeqn` to the sequence number of the last event in the sequence. If your publisher does not know the length of the sequence when it starts

publishing, then `appLastSeqn` need not be set. When your publisher is about to publish the last event of the sequence, `appLastSeqn` could be set to equal `appSeqn`. Setting both `appSeqn` and `appLastSeqn` to -1 indicates the event is empty.

If you want your publisher to have a continuous stream of sequence numbers, then you should use BrokerEvent.setPublishSequenceNumber method to set the appropriate number prior to publishing the event.

Note: The BrokerEvent.setPublishSequenceNumber method does not actually set the `pubSeqn` envelope field. Instead, it specifies the sequence number that the Broker is to use when the event is published by your Broker client.

Read-only Envelope Fields

The table below shows the envelope fields used by the Broker which your application can retrieve, but not alter.

Note: Attempting to retrieve an envelope field that has not been set will cause a BrokerFieldNotFoundException to be thrown.

Envelope Field	Event Editor Type	Description
`age`	`int`	The cumulative time, in seconds, that the event spends on all Brokers. The Broker starts tracking the age of an event when it receives the event from the publishing client. The Broker stops tracking the age of an event when the subscribing client removes the event from the client queue. If the event is routed to successive Brokers, age also includes the length of time the event spends on the other Brokers.
`connectionIntegrity`	`unicode_string`	Indicates whether or not the received event passed over an insecure link.
`destId`	`unicode_string`	Client ID of the event's recipient. This is used only with delivered

Envelope Field	Event Editor Type	Description
		events, described on "Delivering Events" on page 121.
`enqueueTime`	`date`	The date and time that the Broker enqueued the event for the recipient.
`logBroker`	`unicode_string`	The name of the Broker that has this event in its log.
`logHost`	`unicode_string`	The host name and port number of the Broker that has this event in its log.
`pubDistinguishedName`	`unicode_string`	The distinguished name of the Broker client that published the event using an SSL connection.
`pubId`	`unicode_string`	Client ID of the event's publisher. If the publishing client is connect to a different Broker than the recipient, the ID will be fully qualified (prefixed with the name of the publisher's Broker).
`pubNetAddr`	`sequence of bytes`	A sequence of bytes in string format that contains the IP address and port number of the event's publisher. See "The pubNetAddr Envelope Field" on page 75 for more information on this field.
`pubSeqn`	`long`	A 64-bit value representing the event's publish sequence number. The use of publish sequence numbers is described in "Using Sequence Numbers" on page 133.
`pubLabel`	`short[]`	Set by the Broker for an event publish by a client which has an access label.

Envelope Field	Event Editor Type	Description
recvTime	date	The date and time the event was received by the Broker.
route	sequence of structs	See "The route Envelope Field" on page 75 for more information on this field.
uuid	unicode_string	Universally unique identifier assigned to the event. Used to detect duplicate events.

Your application can use the BrokerEvent.get<type>Field methods to obtain the values of an event that it has received. Be sure to use the appropriate method for the envelope field's type.

Note: When referring to envelope fields, you must add `_env.` to each of the field names shown in the table above.

The example below shows how your Broker client can retrieve the `pubSeqn` value of a received event by using the BrokerEvent.getLongField method and specifying the `_env.pubSeqn` field name.

```
. . .
BrokerEvent e;
long seqNumber;
. . .
try {
seqNumber = e.getLongField("_env.pubSeqn");
} catch (BrokerException ex) {
System.out.println("Error on getting envelope field\n"+ex);
return;
  }
. . .
```

The connectionIntegrity Envelope Field

The connectionIntegrity envelope field is a read-only field that describes the integrity of the Broker-to-Broker connections that were used to transport the event.

Value	Meaning
empty string	At some point, the event traveled over a connection that was not encrypted.
"Export"	All the connections used to transport the event had an encryption strength of `ENCRYPT_LEVEL_US_EXPORT` or greater.

Value	Meaning
"US Domestic"	The event traveled exclusively over connections with an encryption strength of `ENCRYPT_LEVEL_US_DOMESTIC`.

Note: The `connectionIntegrity` field is set by the Broker, so it does not reflect the integrity of the connection between the receiving Broker client and its Broker.

The pubNetAddr Envelope Field

This read-only envelope field is set by Broker and contains the IP address and port number of the event's publisher. This field supports IPv6 addresses in the string format.

The example below shows how to retrieve IP address in string format.

```
event.getByteSeqField("_env.pubNetAddr", 0, BrokerEvent.ENTIRE_SEQUENCE).toString()
```

Prior to the string format, the pubNetAddr Envelope field was six-byte long, where the first four bytes represented IPv4 address in the network byte-order, and the last two bytes represented the port in the network byte-order.

To retrieve the IP address in the old six-byte format, set the `pub-net-address-in-deprecated-format` configuration parameter to `1` in the Broker configuration file (awbroker.cfg). By default, `pub-net-address-in-deprecated-format = 0`, where the IP address is retrieved in the string format.

The presence of the pubNetAddr Envelope field is controlled by the Broker configuration file. This envelope field is disabled by default, by you can enable it using the following steps:

1. Edit the Broker configuration file and add the following line:

   ```
   pub-net-address-in-envelope=1
   ```

2. Save the file.
3. Restart the Broker Server.

   Note: This will enable the `pubNetAddr` envelope field for all Brokers within the Broker Server.

4. For complete information on the Broker configuration file, see *Administering webMethods Broker* document.

The route Envelope Field

The `route` read-only envelope field is a sequence of structures that contain event forwarding information. This field is only set on those events which are forwarded from one Broker to another Broker. This envelope field will not be set if both the publishing and receiving clients are both connected to the same Broker. Each structure in the sequence has the format shown in the table below.

Type	Member Name	Description
`string`	`broker`	Name of the Broker.
`date`	`recvTime`	Time the Broker received the event from the publishing client or the other Broker.
`date`	`enqueueTime`	Time the Broker enqueued the event for the next Broker.

The values for `route[0]` will represent the originating Broker. Each time the event is forwarded to another Broker, an additional structure will be added to the `route` sequence.

For more information on Broker-to-Broker communication, see *Administering webMethods Broker*.

Specifying Field Names

Many of the methods offered by BrokerEvent require a field name parameter. How you specify a field name depends on the type of field being referenced and the method being used. The example below shows a hypothetical event type that contains several different types of data fields.

Note: Event field names are case sensitive.

```
Sample::StructEvent {
string title;
float seqA[];
int seqB[][];
struct structA{
BrokerDate date;
int seqC[] [];
}
struct structB{
BrokerDate time;
struct emp {
string name;
string ssn;
}
}
struct structC[] {
BrokerDate date;
int seqD[] [];
}
```

The table below shows the field names you could specify to reference the various fields within the event type shown in the example above.

Field Name	Description	Related BrokerEvent Methods
`title`	The `string` value.	`getStringValue` `setStringValue`
`seqA`	The entire sequence of `float` values.	`getSequenceField` `setSequenceField`
`seqA[0]`	First `float` value in `seqA`	`getFloatField` `setFloatField`
`seqA[]`	Used to obtain the field's type when the number of elements in the sequence is not known.	`getFieldType`
`seqB[0][0]`	First `int` value in `seqB`.	`getIntField` `setIntField`
`seqB[][]`	Used to obtain the field's type when the number of elements in the sequence is not known.	`getFieldType`
`structA`	The entire `structA` structure.	`getStructFieldAsEvent` `setStructFieldFromEvent`
`structA.date`	`BrokerDate` value within `structA`.	`getDateField` `setDateField`
`structA.seqC[0][0]`	First `int` value in `seqC`, within `structA`.	`getIntegerField` `setIntegerField`
`structA.seqC[][]`	Used to obtain the field's type when the number of elements in the sequence is not known.	`getFieldType`
`structB`	The entire `structB` structure.	`getStructFieldAsEvent` `setStructFieldFromEvent`
`structB.time`	`BrokerDate` within `structB`.	`getDateField` `setDateField`

Field Name	Description	Related BrokerEvent Methods
`structB.emp`	The structure within `structB`.	`getStructFieldAsEvent` `setStructFieldFromEvent`
`structB.emp.name`	The `string` within the `emp` structure, within `structB`.	`getStringField` `setStringField`
`structC`	The entire structure sequence `structC`.	`getStructSeqFieldAsEvents` `setStructSeqFieldFromEvents`
`structC[0].date`	`BrokerDate` within the first structure in the `structC` sequence.	`getDateField` `setDateField`
`structC[].date`	Used to obtain the field's type when the number of elements in the sequence `structC` is not known.	`getFieldType`
`structC[0].seqD[0][0]`	First `int` value in `seqD`, within the first structure in the `structC` sequence.	`getIntegerField` `setIntegerField`
`structC[].seqD[][]`	Used to obtain the field's type when the number of elements in the sequences `structC` and `seqD` are not known.	`getFieldType`

5 Subscribing to and Receiving Events

Overview

This chapter describes the webMethods Broker Java API for subscribing to, receiving and processing events. Reading this chapter will help you to understand:

- How to register and cancel event subscriptions.
- How subscription identifiers are created and used with subscriptions.
- How to retrieve events from the Broker using BrokerClient.getEvent or BrokerClient.getEvents.
- How to use the redelivery counter to detect events that your client has received more than once.
- How to retrieve events from the Broker using BrokerClient.poll.
- How to create a subscription that captures deadletters.

Event Subscriptions

In order for your application to be able to receive and process events, it must first create a BrokerClient. Then, any event that is permitted by the client group to which the BrokerClient belongs can be retrieved by your application.

Note: You do not have to register any event subscriptions to receive a *delivered* event. For more information on delivering events, see "Delivering Events" on page 121.

To receive published events, your application must first use the BrokerClient to subscribe to the event types that it wants to receive. Event subscriptions are always made within the context of a particular BrokerClient object.

Note: Event types are defined with Software AG Designer, described in the *Software AG Designer Online Help*. Note that event types are known as *document types* in Designer and other webMethods components.

A single BrokerClient can make multiple subscription requests, so subscriptions are distinguished by the unique combination of the event type name and an optional filter string. Filters, covered in "Using Event Filters" on page 145, allow you to receive only those events that contain certain data in which you are interested. If a BrokerClient requires two subscriptions for the same event type name, a different filter string must be specified for each subscription.

You can use the BrokerClient.getSubscriptions method to obtain the names of all the open subscriptions for a Broker client.

Limits on Subscriptions

The client group to which the BrokerClient object belongs might limit the event types to which the client can subscribe. See "Client Groups" on page 25 for more information. The example below contains an excerpt from a sample application that shows the use of the BrokerClient.canSubscribe method to check for event subscription permission. This method requires a single parameter; an event type name.

```
BrokerClient c;
boolean can_subscribe;
. . .
try {
  can_subscribe = c.canSubscribe("Sample::SimpleEvent");
    catch (BrokerException ex) {
    System.out.println("Error on check for can subscribe\n"+ex);
      return;
}
. . .
```

You can also use the BrokerClient.getCanSubscribeNames method to obtain the names of all the event types to which a Broker client can subscribe.

Subscribing to Events

Your application can subscribe to an event by calling one of the BrokerClient.newOrReconnect methods. The example below contains an excerpt from a sample application that shows the use of the BrokerClient.newSubscription method. This method accepts these parameters:

- An event type name.
- An event filter string or `null` if event filtering is not wanted. Filters are described in "Using Event Filters" on page 145.

```
BrokerClient c;
. . .
/* Make a subscription */
try {
  c.newSubscription("Sample::SimpleEvent",null);
  } catch (BrokerException ex) {
  System.out.println("Error on create subscription\n"+ex);
  return;
}
. . .
```

Uniqueness of Subscriptions

Each subscription that your BrokerClient registers is considered to be unique, based on the event type name and filter specified. After you have registered a subscription for a particular event type name and filter combination, any attempts to register another subscription with the same combination will be ignored.

Note: The BrokerClient.newSubscription method will not throw an exception if you attempt to register the same subscription more than once, it will simply ignore the request.

You can use the BrokerClient.doesSubscriptionExist method to determine whether a subscription has already been registered.

Canceling a Subscription

Your application can cancel an event subscription by calling the BrokerClient.cancelSubscription method. The example below contains an excerpt from a sample application that shows how you can use this method. This method accepts these parameters:

- The event type name, which can optionally contain a wildcard at the end.
- The event filter string that was specified with the original specification. A `null` value can be specified if event filtering was not specified in the original subscription. Event filters are described in "Using Event Filters" on page 145.

```
BrokerClient c;
. . .
/* Cancel the subscription */
try {
  c.cancelSubscription("Sample::SimpleEvent",null);
  } catch (BrokerException ex) {
  System.out.println("Error on canceling subscription\n"+ex);
  return;
}
. . .
```

Using Wildcards

To register or cancel a subscription for multiple event types, you can use a single wildcard character, `*`, at the end of the event type name. To subscribe to all the event types within the scope of `Sample`, you could use the following event type name:

```
Sample::*
```

If you use wildcards in the event type name, the following restrictions apply:

- Only those event types that your client has permission to subscribe to will be included in the subscription.
- Wildcards cannot be used with any event type within the scope of `Broker::Trace` or `Broker::Activity`.
- If a filter string is specified for the subscription, it might not contain any data fields. The filter string might contain envelope fields, however.

Receiving Events in the Get-Events Model

The simplest way to retrieve events is to use the get-events model, which involves these steps:

1. Create a BrokerClient object.
2. Use the BrokerClient to subscribe to one or more event types. If your client only expects to receive delivered events, no subscriptions are necessary.
3. Enter a processing loop in which the BrokerClient.getEvent method is called to retrieve the next event.
4. Extract the fields that you want from each received event using the methods described in "Creating and Initializing Events" on page 55. Return to step 3 and receive the next event.
5. Call BrokerClient.destroy when you are finished.

You can also retrieve events using the poll method or the callback method. For more information, see "Getting Events using the poll Method" on page 91 or "Using the Callback Model" on page 47 respectively.

The example below contains an excerpt from a sample application that shows the use of theBrokerClient.getEvent method. This method accepts a single parameter; the number of milliseconds to wait for an event, if none are currently available. This can be set to -1 if you want to block indefinitely.

```
. . .
BrokerClient c;
BrokerEvent e;
. . .
/* Loop getting events */
count = 1;
while(count <= num_to_receive) {
try {
  e = c.getEvent(-1);
  } catch (BrokerException ex) {
  System.out.println("Error on getting event\n"+ex);
  return;
 }
  ++count;
 }
 . . .
```

Note: The BrokerClient.getEvent method automatically acknowledges all outstanding events for the Broker client. See "Using Sequence Numbers" on page 133 for information on acknowledging events.

Getting Multiple Events

Your application can use the BrokerClient.getEvents method to retrieve up to 160 events with a single call, instead of calling BrokerEvent.getEvent to retrieve events one at a time.

Using the BrokerClient.getEvents method, the subscribing client is given events only once per session. To get events again before acknowledging, disconnect your subscriber and connect again.

The example below shows how you can alter the above sample application to use the BrokerClient.getEvents method. This method accepts these parameters:

- The maximum number of events you want to be returned (up to 160).
- The number of milliseconds to wait for an event, if none are currently available. This can be set to `-1` if you want to block indefinitely.

```
long receive_attempts = 10;
long number_received;
BrokerClient c;
BrokerEvent e[];
. . .
/* Loop getting events */
count = 1;
while(count <= num_to_receive) {
  try {
  e = c.getEvents(20, -1);
  } catch (BrokerException ex) {
  System.out.println("Error on getting events\n"+ex);
  return;
  }
  /* process the received events */
  . . .
  ++count;
  }
  . . .
```

Note: The BrokerClient.getEvents method automatically acknowledges all outstanding events for the Broker client. See "Using Sequence Numbers" on page 133 for information on acknowledging events.

Synchronous Get Events

To specify a synchronous get events using any of the BrokerClient.getEvent or BrokerClient.getEvents methods, you must use the BrokerClient.SYNCHRONOUS definition as the timeout value for the methods.

If the Broker receives the "events" protocol with its flag set to "synchronous" then the Broker will either:

- Return any available events up to the maximum number of requested events
- Immediately return indicating that no events are available.

  When there are no events available, the Broker will not submit a pending request for events on behalf of the client.

If the Broker API requests a synchronous get events, one of the following can occur:

- If the Broker does not return with a response within the default Broker timeout, then a BrokerTimeoutException will be thrown.

- If the Broker returns with events, then the events are returned by the get events method.
- If the Broker returns an indication that no events are available then the Broker client's get events methods will return both of the following:
  - A null for the Brokerclient.getevent methods
  - A zero-length BrokerEvent array for the BrokerClient.getEvents methods

The example below shows the use of synchronous get events using the BrokerClient.getEvent and BrokerClient.getEvents methods. To specify synchronous get events, set the timeout value to the BrokerClient.SYNCHRONOUS.

```
// Example 1
BrokerClient client;
BrokerEvent event = client.getEvent(BrokerClient.SYNCHRONOUS);
if (event != null) {
process(event);
}

// Example 2
BrokerClient client;
BrokerEvent[] events = client.getEvents(10, BrokerClient.SYNCHRONOUS);
if (events.length > 0) {
  process(events);
}
```

For more information, see the BrokerClient.getEvent and BrokerClient.getEvents methods.

Detecting Redelivered Events

When you enable the Broker's redelivery counting feature in the connection descriptor, your client can detect whether it has received an instance of a particular event more than once. A redelivery is indicated if an event's redelivery count is greater than zero.

In the BrokerConnectionDescriptor, you enable the redelivery counter, and use it in one of two modes, *manual* or *automatic*. To able the redelivery counter, you use the setRedeliveryCountEnabled method. The default mode is manual. In this mode your client must explicitly increment the redelivery counter.

To enable automatic mode, call setAutomaticRedeliveryCount after you call setRedeliveryCountEnabled. In automatic mode, the Broker automatically increments the redelivery counter when it re-sends an event to a client.

In either mode, the Broker sets the counter to zero the first time it sends an instance of an event to the client.

The following table summarizes the methods associated with the redelivery feature:

Method	Description
`BrokerConnectionDescriptor.setRedeliveryCountEnabled`	Enables the redelivery counter in the connection descriptor.

Method	Description
`BrokerConnectionDescriptor.getRedeliveryCountEnabled`	Returns the current value of the redelivery counting option from the connection descriptor.
`BrokerConnectionDescriptor.setAutomaticRedeliveryCount`	Sets the redelivery counting option to automatic mode. The default value is manual mode.
`BrokerConnectionDescriptor.getAutomaticRedeliveryCount`	Returns the current value of the automatic redelivery option from the connection descriptor. You use this method to determine whether the redelivery counter is configured for automatic or manual mode.
`BrokerEvent.getRedeliveryCount`	Returns the current value of the redelivery counter.
`BrokerClient.incrementRedeliveryCount`	Increments the redelivery counter by one.

Note: The redelivery counter applies only to guaranteed events. Volatile events always have a redelivery count of `-1`. Additionally, if a client does not enable redelivery counting, all events that it receives will have a redelivery count of `-1`.

Manual Redelivery Counting

When you enable manual counting mode, your client must explicitly update the counter by calling BrokerClient.incrementRedeliveryCount after it successfully receives an event from the Broker. (Only the client session that receives an event can update the redelivery count for that event.)

After incrementing the counter, your client must immediately dispatch the event and the counter value its user code. The user code must accept the redelivery count (an integer) and take appropriate steps to process the event if the value of the counter is greater than zero.

The following snippet illustrates how you would count redeliveries in manual mode.

```
static String broker_host = "localhost";
static String broker_name = null;
static String client_group = "sample";

BrokerConnectionDescriptor d;
BrokerClient c;
BrokerEvent e;
long rsn;
```

```
boolean rd_mode = true;
int rd_count;
. . .
/* -------------------------------------------------*/
/* Create descriptor and enable Manual Redelivery */

try {
d = new BrokerConnectionDescriptor;
d.setRedeliveryCountEnabled(rd_mode);
} catch (BrokerException ex) {
        System.out.println("Error on create descriptor\n"+ex);
        return;
}
/* ------------------------------------------------ */
/* Create client using connection descriptor       */

try {
 c = new BrokerClient(broker_host, broker_name,
                           null, client_group,
                           "Subscriber Sample #1",d);
} catch (BrokerException ex) {
        System.out.println("Error on create client\n"+ex);
        return;
}
. . .
/* -------------------------------------------------*/
/* Get event, update count, pass event to user code */

while (dispatchingDocuments() == true) {
e = BrokerClient.getEvent();
rsn = e.getReceiptSequenceNumber();
rd_count = e.getRedeliveredCount();
if (c.incrementRedeliveryCount (rsn) != OK)
break;
if (dispatchToUserCode(e, rd_count) == OK) {
c.acknowledge(rsn);
}
}
 . . .
```

Automatic Redelivery Counting

If the client is using automatic redelivery counting mode, the Broker automatically updates the redelivery counter when delivers an event to a client. As in manual mode, the redelivery counter is set to zero the first time the event is delivered to a client. If the Broker resends the event to the client, it increments the event's redelivery counter before sending the event out.

The way in which a client uses automatic redelivery counting mode is nearly the same as it would use manual mode. As you can see by the following snippet, the code is identical to the manual mode example on above in "Manual Redelivery Counting" on page 86, except that 1) the client sets the connection descriptor automatic mode and 2) it omits the call to the incrementRedeliveryCount method.

```
static String broker_host = "localhost";
static String broker_name = null;
static String client_group = "sample";

BrokerConnectionDescriptor d;
BrokerClient c;
```

```
BrokerEvent e;
long rsn;
boolean rd_mode = true;
int rd_count;
. . .
/* -------------------------------------------------*/
/* Create descriptor and enable Automatic Redelivery */

try {
d = new BrokerConnectionDescriptor;
d.setAutomaticRedeliveryCount(rd_mode);
} catch (BrokerException ex) {
        System.out.println("Error on create descriptor\n"+ex);
        return;
}
/* ------------------------------------------------- */
/* Create client using connection descriptor       */

try {
 c = new BrokerClient(broker_host, broker_name,
                           null, client_group,
                           "Subscriber Sample #1",d);
} catch (BrokerException ex) {
        System.out.println("Error on create client\n"+ex);
        return;
}
. . .
/* -------------------------------------------------*/
/* Get event, update count, pass event to user code */

while (dispatchingDocuments() == true) {
e = BrokerClient.getEvent();
rsn = e.getReceiptSequenceNumber();
rd_count = e.getRedeliveredCount();
if (dispatchToUserCode(e, rd_count) == OK) {
c.acknowledge(rsn);
}
}
. . .
```

As shown above, your Broker client dispatches the event and the redelivery count to user code. The user code must accept the redelivery count (an integer) and take appropriate steps to process the event if the value of the counter is greater than zero.

Redelivery and Older Versions of webMethods Software

If a client attempts to enable redelivery counting while running against a Broker running a versions prior to 6.1, the client will receive a BrokerInsufficientVersionException when it attempts to create a BrokerClient.

Subscription Identifiers

The webMethods Broker API allows you to associate an arbitrary value, called a subscription identifier, with an event subscription. You can use subscription identifiers to quickly determine how a retrieved event is to be processed. You might also find it helpful to use subscription identifiers if your application plans to use the callback model

for retrieving and processing events. The callback model is described in "Using the Callback Model" on page 47.

Subscription IDs do not uniquely identify a particular subscription, so you can create different subscriptions and with the same subscription ID.

Note: You cannot register the same subscription with more than one subscription ID. This means that if you register a particular subscription for a BrokerClient and assign it a subscription ID of 1, any attempt to register the same subscription with a different subscription ID will be ignored. See "Uniqueness of Subscriptions" on page 81 for more information on what differentiates one subscriptions from another.

Specifying Subscription IDs

The example below illustrates the use of the BrokerClient.newSubscription method to associate two subscription identifiers with two different subscriptions. This version of the BrokerClient.newSubscription method accepts these parameters:

- The subscription identifier.
- An event type name.
- An event filter string or `null` if event filtering is not wanted. Filters are described in "Using Event Filters" on page 145.

Assigning subscription identifiers

```
. . .
/* Make subscription #1*/
try {
c.newSubscription( 1, "Sample::SimpleEvent1",null);
} catch (BrokerException ex) {
System.out.println("Error on create subscription #1\n"+ex);
return;
 }
/* Make subscription #2*/
try {
c.newSubscription( 2, "Sample::SimpleEvent2",null);
} catch (BrokerException ex) {
System.out.println("Error on create subscription #2\n"+ex);
return;
 }
. . .
```

Obtaining Subscription Identifiers

The example below shows how you could use the BrokerEvent.getSubscriptionIds method to obtain the subscription identifier of a retrieved event.

Note: If the filter string specified for two or more subscriptions have overlapping criteria, it is possible for a retrieved event to be associated with more than one subscription identifier. For more information on event filters, see "Using Event Filters" on page 145.

This example assumes that the subscriptions described in the example in "Specifying Subscription IDs" on page 89 have already been made and that there is only one subscription identifier associated with any retrieved event. The BrokerEvent.getSubscriptionIds method accepts no parameters.

Note: Events that are delivered to your Broker client do not have subscriptions or subscription IDs. For more information on delivering events, see "Delivering Events" on page 121.

```
. . .
BrokerClient c;
BrokerEvent e;
int subscriptions[];
...
while(1) {
  try {
e = c.getEvent(-1);
} catch (BrokerException ex) {
System.out.println("Error on getting event\n"+ex);
return;
}
/* get the event's subscription identifier */
subscriptions = e.getSubscriptionIds();
if((subscriptions != null) && (subscriptions.length > 0)){
switch( subscriptions[0] ) {
case 1:
/* process Sample::SimpleEvent1 . . . */
break;
case 2:
/* process Sample::SimpleEvent2 . . .*/
break;
default:
break;
}
}
}
 . . .
```

Generating Subscription Identifiers

You application can generate subscription identifiers itself or it can call one of the following webMethods Broker API methods to generate a subscription identifier.

The BrokerClient.makeSubId method creates a subscription identifier for a specified client. The first time it is called for a particular BrokerClient, it will return a value of 1. The second time it is called for the same BrokerClient, it will return a value of 2, and so on. The subscription identifiers are not guaranteed to be unique if your application disconnects and reconnects the BrokerClient, as described in "Disconnecting and Reconnecting" on page 29.

You can use the BrokerClient.makeSubId method to create unique subscription identifiers for Broker clients that you intend to disconnect and reconnect. This method contacts the Broker and looks up all the existing subscription identifiers in use by your application. It then assigns a subscription identifier that is guaranteed to be unique for all Broker clients that you might use. Another important feature is that after calling this method

once, all subsequent calls to BrokerClient.makeSubId will return unique subscription identifiers.

BrokerSubscription Objects

The BrokerSubscription object provides a convenient way to refer to and manage subscriptions by defining the following three subscription attributes as data members:

- The event's subscription identifier.
- The event's name.
- An event filter string, or `null` if event filtering is not wanted. Filters are described in "Using Event Filters" on page 145.

BrokerSubscription objects are used by the following webMethods Broker API methods:

- BrokerClient.cancelSubscription method.
- BrokerClient.cancelSubscriptions method.
- BrokerClient.getSubscriptions method.
- BrokerClient.newOrReconnect method.
- BrokerClient.newSubscriptions method.

Getting Events using the poll Method

You can use the BrokerClient.poll method to simultaneously poll the queues of multiple Broker clients with a single thread. Unlike the "get events" model, where clients check their queues in serial fashion and block for a specified interval if their queue is empty, the poll method enables an application to dispatch a client to its queue only if there are events there for it to retrieve. This approach results in a more efficient process that wastes fewer cycles on non-productive work (i.e., blocking while a client attempts to retrieve events from an empty queue).

Because the poll method uses a single thread to monitor the queues of multiple clients, it is also easier to implement than spawning individual threads for each client.

The BrokerClientPoll Object

A BrokerClientPoll object maintains information that the polling process takes as input and returns as output. To use the polling model, an application must create a BrokerClientPoll object for each Broker client whose queue the application will poll.

When an application instantiates a BrokerClientPoll object, it must provide two parameters to the constructor, 1) the BrokerClient whose queue the application wants to poll and 2) the `GET_EVENTS` flag, which tells the poll method to poll for events.

Note: Despite its name, the `GET_EVENTS` flag does not cause the poll method to actually retrieve events from a client's queue. Your application code must do that. The poll method merely indicates that events are available and that the getEvents method will not block if executed.

Key Methods used in the Polling Model

The following table summarizes the primary methods and constructors associated with the polling model. For a complete list, see BrokerClientPoll.

Method/Constructor	Description
`BrokerClientPoll`	The constructor that generates a BrokerClientPoll object for a specified Broker client.
`BrokerClient.poll`	Polls the Broker for events in queues belonging to a specified set of Broker clients, and returns the BrokerClientPoll object for each client that has one or more events in its queue.
`BrokerClientPoll.getBrokerClient`	Returns the `BrokerClient` associated with a BrokerClientPoll object.

Using the poll Method

The following procedure describes the general steps you use the poll method to retrieve events.

1. Create the Broker clients (e.g., BrokerClient[]) whose queues you want to poll.
2. Generate an array containing a BrokerClientPoll object for each client in BrokerClient[].
3. Call the BrokerClient.poll method and pass in the BrokerClientPoll[] and a time-out interval (in milliseconds). This method will check the queue of each client represented BrokerClientPoll[]. If the method discovers at least one event in a client's queue, it sets a flag in that client's BrokerClientPoll object. If the method does not find an event in any of the client queues, it continues to poll until 1) one or more events arrive, 2) the specified timeout interval expires, 3) the thread is interrupted, whichever occurs first.
4. Loop through the BrokerClientPoll[] that the poll method returns and retrieve the events from the queues of the clients represented in that array. (The array *will not* contain clients whose queues were empty.)

Important: The queues of the clients in the array have already been primed using `prime(1)`. Your application code does not need to re-prime the queues unless you want it to fetch more than a single event.

A Polling Example

The following snippet illustrates how you would code the steps described above.

```
static String broker_host = "localhost";
static String broker_name = null;
static String client_group = "sample";
int N = numberOfClients;

. . .
/* ----------------------------------------------------------*/
/* Allocate the array for BrokerClients                     */
BrokerClient[] c = new BrokerClient[N]

/* ----------------------------------------------------------*/
/* Create BrokerClients                                     */
   . . .

/* ----------------------------------------------------------*/
/* Allocate array for BrokerClientPoll Objects               */

BrokerClientPoll[] poll = new BrokerClientPoll[N];

/* ----------------------------------------------------------*/
/* Create BrokerClientPoll objects for each BrokerClient    */

for (int i = 0; i < N; i++) {
 poll[i] = new BrokerClientPoll(c[i], BrokerClientPoll.GET_EVENTS, null);

       /* ----- Prime the client -------------------------- */
c[i].prime(NUM_TO_PRIME);
}

/* ----------------------------------------------------------*/
/* Poll the queues of clients identified in BrokerClientPoll[]

   for (;;) {
BrokerClientPoll[] result = BrokerClient.poll(poll, -1);

/* ----------------------------------------------------------*/
/* Get events for BrokerClients identified in results array  */

for (int i = 0; i < result.length; i++) {
if (result[i].isReady(BrokerClient.GET_EVENTS)) {
 BrokerEvents[] events = result[i].getBrokerClient().getEvents(MAX, 0);

/* ----- Re-prime the client ---------------- */
result[i].getBrokerClient().prime(NUM_TO_PRIME);
}
}
 }
```

Detecting Deadletters

If a Broker receives an event for which it has no subscribers, it discards the event *unless* a dead letter subscription exists for that event type. A dead letter subscription enables you to trap events for which no subscriptions exist. Among other things, trapping dead letters can help to quickly identify and resolve discrepancies between a published event and the filtering criteria specified by a subscriber.

Creating a Deadletter Subscription

To subscribe to dead letters, you use the BrokerClient.newSubscription method. In the *filter* parameter for this method, you specify the DeadLetterOnly hint.

As shown in the following examples, you can use wildcards to specify the types of events for which you want to receive dead letters.

The following code fragment shows how you would create a dead letter subscription for a specific event type.

```
BrokerClient c;
. . .
/* Create a deadletter subscription for an event type*/
try {
c.newSubscription("APITest::SimpleEvent","{hint:DeadLetterOnly}})";
catch (BrokerException ex) {
System.out.println("Error on create subscription\n"+ex);
return;
}
. . .
```

The following code fragment show how you would use a wildcard character to create a dead letter subscription to all events in a particular namespace.

```
BrokerClient c;
. . .
/* Create a deadletter subscription for all events in a namespace*/
try {
c.newSubscription("APITest::*","{hint:DeadLetterOnly}})";
catch (BrokerException ex) {
System.out.println("Error on create subscription\n"+ex);
return;
}
. . .
```

The following code fragment show how you would use the wildcard character to create a dead letter subscription for all events.

```
BrokerClient c;
. . .
/* Create a deadletter subscription for all event types*/
try {
c.newSubscription("*","{hint:DeadLetterOnly}})";
catch (BrokerException ex) {
System.out.println("Error on create subscription\n"+ex);
return;
}
```

. . .

Dead Letter Subscriptions and Filters

Be aware that when you create a dead letter subscription, you are subscribing to all events of the specified type(s). You cannot filter events in a dead letter subscription. If you specify the DeadLetterOnly hint and a filter in the same subscription, the filter is ignored.

Dead Letter Permissions and Persistence

With respect to permissions and persistence, the following points apply to dead letter subscriptions:

- A dead letter subscription is subject to the same permission requirements as a regular subscription. That is, a client can receive dead letters for only the events on its can-subscribe list. If a client specifies an event type using wildcard characters, that client will receive only those dead letters that coincide with its can-subscribe list.
- Persistence of dead letter events is determined by the normal rules of client types and event types.
- Dead letter subscriptions apply to both published and delivered events. If the Broker receives a delivered event that has been addressed to a non-existent client, the Broker will deposit that event in the dead letter queue if one exists for the event type.

How Dead Letter Subscriptions Relate to Regular Subscriptions

The following describes the way in which dead letter subscriptions are affected by regular subscriptions for the same event type.

- If the Broker permits identical subscriptions to a regular event and a dead letter event. In this situation, the dead letter subscribers simply never receive any events.
- If a wildcard definition in a dead letter subscription overlaps with that of a regular subscription, the Broker distributes events that match the regular subscriptions to the appropriate clients and then routes the others to the dead letter subscriber.
- An event type can have more than one dead letter subscriber.

Dead Letter Behavior in a Territory

The following points describe the behavior of dead letter subscriptions within a territory.

- Dead letter subscriptions are implicitly "local-only," meaning that dead letter events published by one Broker are not forwarded to dead letter subscribers on other Brokers.

- To ensure that an instance of an event can be captured as a dead letter by any Broker to which it might be forwarded, you need to create a dead letter subscription for that event type on each Broker in the territory.

6 Using Request-Reply

Overview

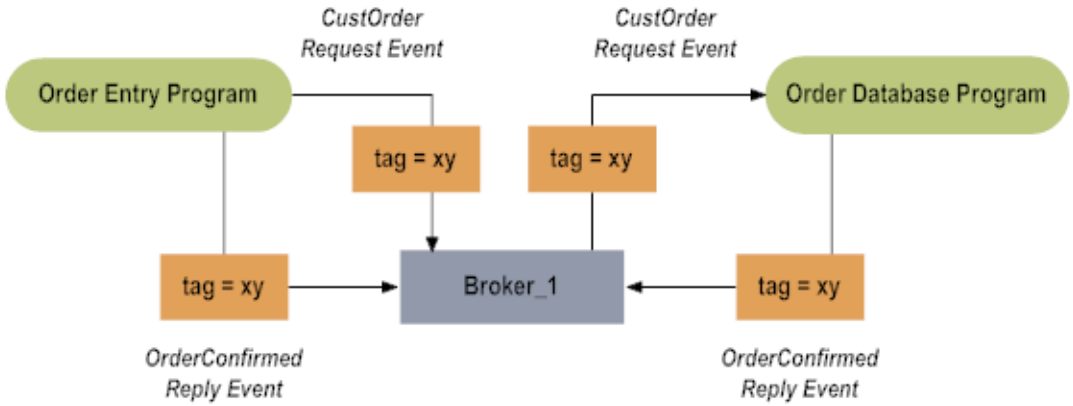
This chapter describes the webMethods Broker Java API for implementing applications that use the request-reply event processing model. Reading this chapter will help you to understand:

- The use of event tag fields to match request events that are sent with reply events that are received.
- The various types of reply events.
- Using get-event, callback, and the BrokerClient.publishRequestAndWait approaches in implementing a requestor application.
- Using the reply methods to implement a request server application.

The Request-Reply Model

You can use the request-reply model for applications that publish or deliver a request event to a server application which is expected to return a reply event. The reply event can contain data or can simply be an acknowledgment with no data.

Request and reply events and their tag fields



Request Events and the Tag Field

A `tag` envelope field is set by your requestor application to identify the request event that it is sending. When a server application receives the request event and prepares the reply event, it ensures that the same tag field is set for the reply as was received on the request event. If your requestor sends several different request events, it should set each with a different tag field. When your requestor receives a reply event, it can check the tag field to determine the original request with which the reply is associated.

Getting and Setting the Tag

Use the BrokerEvent.getTag method to obtain the `tag` field from an event. Use the BrokerEvent.setTag method to set an event's `tag` field.

Using the trackId Field

The `trackId` envelope field can be set by a publishing client application to a unique identifier that will allow the event to be tracked. The use of this envelope field allows an event that is received and then re-published to still be tracked. This envelope field also allows you to track several events that can be associated with a single logical transaction.

Note: If `trackId` envelope field is not set, any of the Broker Server methods for delivering reply events will automatically set it to the value contained the `pubId` envelope field.

Reply Events

The content of a reply events can vary, depending on the design of the requestor and server applications. The request event's infoset defines the reply event that is expected.

A reply event can take one of the following forms:

- A null event that has no data.
- An acknowledgment that indicates the operation was successful.
- A single reply event containing application-defined data.
- A series of reply events containing application-defined data.
- An error reply.

If a series of reply events are returned, the `appSeqn` and `appLastSeqn` envelope fields can be accessed to determine the event's sequence position. See "Envelope Fields" on page 69 for more information.

Determining a Reply Event's Type

The following methods are provided to help your client application determine what type of reply event has been received.

- Use the BrokerEvent.isAckReply method to determine whether an event is an acknowledgment reply.
- Use the BrokerEvent.isErrorReply method to determine whether an event is an error reply.
- Use the BrokerEvent.isLastReply method to determine whether an event is the last reply in a sequence.

- Use the BrokerEvent.isNullReply method to determine whether an event is a null reply.

The Requestor

You have several processing models to choose from when designing a requestor application. These design alternatives include:

- Use BrokerClient.publish to send the request event, use BrokerClient.getEvent to receive possible reply events, and check each event received for a matching tag.
- Create a callback object for the response event and register it with BrokerClient.registerCallbackWithTag, specifying the tag you will use. Use BrokerClient.publish to send the request event and then use one of the Broker Server dispatching methods to dispatch received events.
- Use BrokerClient.publishRequestAndWait to send the request event and wait until a reply event is received.

The following example shows the code that implements the following preliminary steps:

1. Create a BrokerClient object.
2. Check for publication permission for the request event type.
3. Check for subscription permission for the reply event type. This is done even though the reply event will be delivered because it indicates if the requestor's client group will allow it to receive the reply event type.
4. Create a request BrokerEvent object.
5. Create a tag for the request event using the BrokerClient.makeTag method.
6. Set the request event's tag field using the BrokerEvent.setEventTag method.

```
. . .
BrokerClient c;
BrokerEvent e;
boolean permission;
int request_tag;
. . .
/* Create a client */
try {
c = new BrokerClient(broker_host, broker_name, null,
client_group, "Publish Sample #1",null);
} catch (BrokerException ex) {
. . .
}
/* Check if can subscribe */
try {
permission = c.canSubscribe("Sample::Reply");
} catch (BrokerException ex) {
. . .
}
/* Check if can publish */
try {
permission = c.canPublish("Sample::Request");
} catch (BrokerException ex) {
```

```
. . .
}
/* Create the request event */
try {
e = new BrokerEvent(c, "Sample::Request");
} catch (BrokerException ex) {
. . .
}
/* Create tag and set request event's tag field */
request_tag = c.makeTag();
try {
e.setTag(tag);
} catch (BrokerException ex) {
. . .
}
. . .
```

Using the Get-event Approach

After the preliminary processing described in the example above in "The Requestor" on page 100 has been completed, the get-event design involves these steps:

1. Use BrokerClient.publish to publish the request event.
2. Enter a processing loop and receive events with BrokerClient.getEvent.
3. Use BrokerEvent.getTag to check each received event for a tag that matches the request event's tag.

   The following example illustrates receiving a reply event with BrokerClient.getEvent:

```
. . .
BrokerClient c;
BrokerEvent e;
boolean done;
int received_tag;
long request_tag;
. . .
/* Create BrokerClient, check subscription and publish permissions,
 * create request BrokerEvent, create tag, set tag field
 */
. . .
/* Publish the request */
try {
c.publish(e);
} catch (BrokerException ex) {
. . .
}
/* Loop getting events */
done = 0;
while(!done) {
try { /* get an event */
e = c.getEvent();
} catch (BrokerException ex) {
. . .
}
try { /* get the event's tag */
received_tag = e.getTag();
} catch (BrokerException ex) {
. . .
}
```

```
/* see if event matches request */
if (request_tag == received_tag) {
if (e.isNullReply())
/* handle null reply */
} else if (e.isErrorReply()) {
/* handle error reply */
} else {
/* process the event */
. . .
}
done = e.isLastReply();
}
}
. . .
```

Callbacks with Tags

After the preliminary processing described in the example above in "The Requestor" on page 100 has been completed, the callback design involves these steps:

1. Use BrokerClient.publish to publish the request event.
2. Use BrokerClient.registerCallback to register a general callback object.
3. Use BrokerClient.registerCallbackForTag to register a specific callback object for the reply event with the request event's tag.
4. Use one of the callback dispatching methods, described on "Dispatching Callback Methods" on page 52, to receive events and dispatch the appropriate callback object's event handling method.

   The following example illustrates receiving a reply event with a callback object:

```
public static void main(String args[])
{
SampleCallback1 general_callback =
new SampleCallback1(num_to_receive);
SampleCallback2 specific_callback =
new SampleCallback2(num_to_receive);
. . .
/* Publish the request */
try {
c.publish(e);
} catch (BrokerException ex) {
. . .
}
/* Register general callback */
try {
c.registerCallback(general_callback,null);
} catch (BrokerException ex) {
System.out.println("Error on registering general callback\n"+ex);
return;
}
/* Register specific callback for the request_tag */
try {
c.registerCallbackForTag(request_tag, true,
specific_callback,null);
} catch (BrokerException ex) {
System.out.println("Error on registering specific callback\n"+ex);
return;
}
```

```
/* Dispatch loop */
try {
BrokerClient.dispatch(-1);
} catch (BrokerException ex) {
System.out.println("Error on dispatch\n"+ex);
return;
}
. . .
```

The following example illustrates the callback implementation:

```
public class SampleCallback2 implements BrokerCallback
{
. . .
/* Method to handle the webMethods Broker Server event callbacks. */
public boolean handleBrokerEvent(
BrokerClient client,
BrokerEvent event,
Object client_data)
{
if (event.isNullReply()) {
/* handle null reply */
System.out.println("Null reply received.\n");
} else if (event.isErrorReply()) {
/* handle error reply */
System.out.println("Error reply received.\n");
} else {
/* process the event */
. . .
}
return true;
}
}
```

Using publishRequestAndWait

After the preliminary processing described in the example above in "The Requestor" on page 100 has been completed, the callback approach involves these steps:

1. Use BrokerClient.registerCallback to register a general callback object.
2. Use BrokerClient.publishRequestAndWait to publish the request and wait for the reply. You can also use the BrokerClient.deliverRequestAndWait method if you want to send the request event to a specific Broker client.

> **Note:** No event subscription is necessary in this model because the reply event will be delivered, not published.

The following example illustrates how to use BrokerClient.publishRequestAndWait:

```
public static void main(String args[])
{
BrokerClient c;
BrokerEvent e, events[];
int i;
SampleCallback1 general_callback =
new SampleCallback1(num_to_receive);
. . .
/* Register general callback */
try {
```

```
 c.registerCallback(general_callback,null);
 } catch (BrokerException ex) {
 . . .
 }
 /* publish the request and wait for a reply */
 try {
 events[] = c.publishRequestAndWait(e, 6000);
 } catch (BrokerException ex) {
 . . .
}
/* Process received events */
for( i = 0; i < events.length; i++) {
if (events[i].isNullReply()) {
/* process null reply */
} else if (events[i].isErrorReply()) {
/* process error reply */
} else {
/* process reply */
}
}
. . .
```

The Server

Server applications must be allowed to subscribe to all of the event types that are to be processed. In response to a request event, servers must also be prepared to deliver a reply event of the expected type. Generally, your server application should follow these steps:

1. Check for permission to subscribe to the request event types.
2. Check for the appropriate reply event publishing permissions.
3. Retrieve an event and process it.
4. Deliver the appropriate reply event(s).

Checking Subscription and Publishing Permissions

The Server shown in the following example checks to see if it has permission to subscribe to the request event and to deliver the appropriate reply event. The BrokerClient.canPublish method is used to determine whether the Broker client has the necessary permissions to deliver the various reply events. In this example, the application expects that it might have to deliver the event types shown in the following table:

Event Type	Description
`Adapter::error`	Used to indicate that an exception occurred in the processing of the event.

Event Type	Description
`Adapter::ack`	Used if the infoset for `Sample::Request` specifies that a simple success or failure indication is all that is expected.
`Sample::Reply`	Used if the infoset for `Sample::Request` defines a specific event type as a response.

```
boolean permission;
BrokerClient c;
. . .
/* Create a Broker client */
. . .
/* Check if can publish reply */
try {
permission = c.canPublish("Sample::Reply");
} catch (BrokerException ex) {
. . .
}
if(!permission) {
System.out.println("Permission to publish Sample::Reply denied.\n");
return;
}
/* Check if can publish error reply */
try {
permission = c.canPublish("Adapter::error");
} catch (BrokerException ex) {
. . .
}
if(!permission) {
System.out.println("Permission to publish Adapter::error denied.\n");
return;
}
/* Check if can publish acknowledgement */
try {
permission = c.canPublish("Adapter::ack");
} catch (BrokerException ex) {
. . .
}
if(!permission) {
System.out.println("Permission to publish Adapter::ack denied.\n");
return;
}
/* Check if can subscribe to the request event */
try {
permission = c.canSubscribe("Sample::Request");
} catch (BrokerException ex) {
. . .
}
if(!permission) {
System.out.println("Cannot subscribe to Sample::Request denied.\n");
return;
}
/* Subscribe to the request event */
try {
c.newsubscription("Sample::request", null);
} catch (BrokerException ex) {
. . .
}
. . .
```

Processing Request Events

Your server application has all of the usual options for receiving and processing events. It can use the BrokerClient.getEvent method to receive events within a manually coded loop, described in Chapter 4. If your server subscribes to several request event types, it can use the callback model described in "Using the Callback Model" on page 47.

The server application can also associate an identifier with each of the subscriptions that it registers. When an event is received, the server application can easily determine how to process the request event. See "Subscription Identifiers" on page 88 for more information.

The following example shows an excerpt of a server application that uses BrokerClient.getEvent method to receive an event and determines if it is a request event.

```
. . .
BrokerClient c;
BrokerEvent e;
int n, count = 200;
String event_type_name;
. . .
/* Loop getting events */
n = 1;
while(n < count) {
try {
e = c.getEvent(-1);
} catch (BrokerException ex) {
. . .
}
/* Check if it is a request */
try {
event_type_name = e.getEventTypeName();
} catch (BrokerException ex) {
. . .
}
if ((event_type_name != null) &&
(event_type_name.equals("Sample::Request")) {
/* Process the request (see following excerpts) */
. . .
```

Delivering Replies

After you have processed a request event, you can use one of several methods to deliver the appropriate type of reply event.

Note: All of the following event delivery methods determine the destination identifier based on the `pubId` field of the original request event. None of these methods will return an error if the destination identifier refers to a Broker client that no longer exists.

Delivering Acknowledgment Replies

Your server application can deliver an acknowledgment reply if the infoset for the request event type defines that a simple success or failure indication is all that is required. When invoking the BrokerClient.deliverAckReplyEvent method, you can either specify a valid publish sequence number or specify a value of zero if you do not want to use publish sequence numbers. For more information, see "Using Sequence Numbers" on page 133.

```
. . .
try {
c.deliverAckReplyEvent(e,0);
} catch (BrokerException ex) {
...
}
 . .
```

Delivering Error Replies

The server application delivers an error reply to indicate that some sort of exception occurred while attempting to process the request event.

```
. . .
BrokerEvent err[] = new BrokerEvent[1];
. . .
/* Make error reply event */
err[0] = new BrokerEvent(c,"Adapter::error");
try {
c.deliverReplyEvent(e, err);
} catch (BrokerException ex) {
. . .
}
. . .
```

Delivering Null Replies

The server application can deliver a null reply if the request was successfully processed and there is no data to be returned. When invoking the BrokerClient.deliverNullReplyEvent method, you can either specify a valid publish sequence number or specify a value of zero if you do not want to use publish sequence numbers. For more information, see "Using Sequence Numbers" on page 133.

```
. . .
try {
c.deliverNullReplyEvent(e,"Sample::Reply",0);
} catch (BrokerException ex) {
. . .
}
. . .
```

Delivering One or More Reply Events

The server application can deliver one or more reply events in response to a single request event, as shown in the following example.

```
. . .
BrokerEvent replies[];
```

```
. . .
/* prepare the reply events */
. . .
/* deliver the replies */
try {
c.deliverReplyEvents(e, replies);
} catch (BrokerException ex) {
. . .
}
. . .
```

Delivering Partial Replies

Your server application might need to deliver multiple reply events to satisfy a request event that it receives. If all of the reply event data cannot be read into memory or is not immediately available, the server can choose to send the reply events in batches rather than all at once.

The BrokerClient.deliverPartialReplyEvents method can be used in these cases. You must set the `flag` parameter to `REPLY_FLAG_START`, `REPLY_FLAG_CONTINUE`, or `REPLY_FLAG_END` to indicate the start, continuation, and end of the reply event sequence.

If all of the replies can be sent in one batch, simply set the `flag` to `REPLY_FLAG_START_AND_END`.

```
BrokerEvent reply_events[];
int token, flag;
boolean done = false;
. . .
/* prepare some reply events */
. . .
flag = BrokerClient.REPLY_FLAG_START;
while(!done) {
try {
token = c.deliverPartialReplyEvents(e, reply_events,
flag, token);
} catch (BrokerException ex) {
...
}
/* prepare more reply events */
. . .
flag = BrokerClient.REPLY_FLAG_CONTINUE;
}
flag = BrokerClient.REPLY_FLAG_END;
/* prepare the last reply event */
. . .
try {
token = c.deliverPartialReplyEvents(e, reply_events,
flag, token);
} catch (BrokerException ex) {
. . .
}
. . .
```

7 Transaction Semantics

Overview

This chapter describes the webMethods Broker Java API for implementing applications that publish events using a transaction processing model. Reading this chapter will help you to understand transactional client processing with Broker.

About Transactional Client Processing

Transactional client processing facilitates publish, acknowledge, and get event functions within the context of a transaction.

Transaction processing allows your Broker client to group the events it publishes as a single unit of work called a transaction. A transaction either completes successfully, is rolled back to some known earlier state, or it fails. After all of the events that make up a transaction have been published, your Broker client then ends the transaction. A transaction can be ended by committing the transaction or aborting the transaction.

About the BrokerTransactionalClient Class

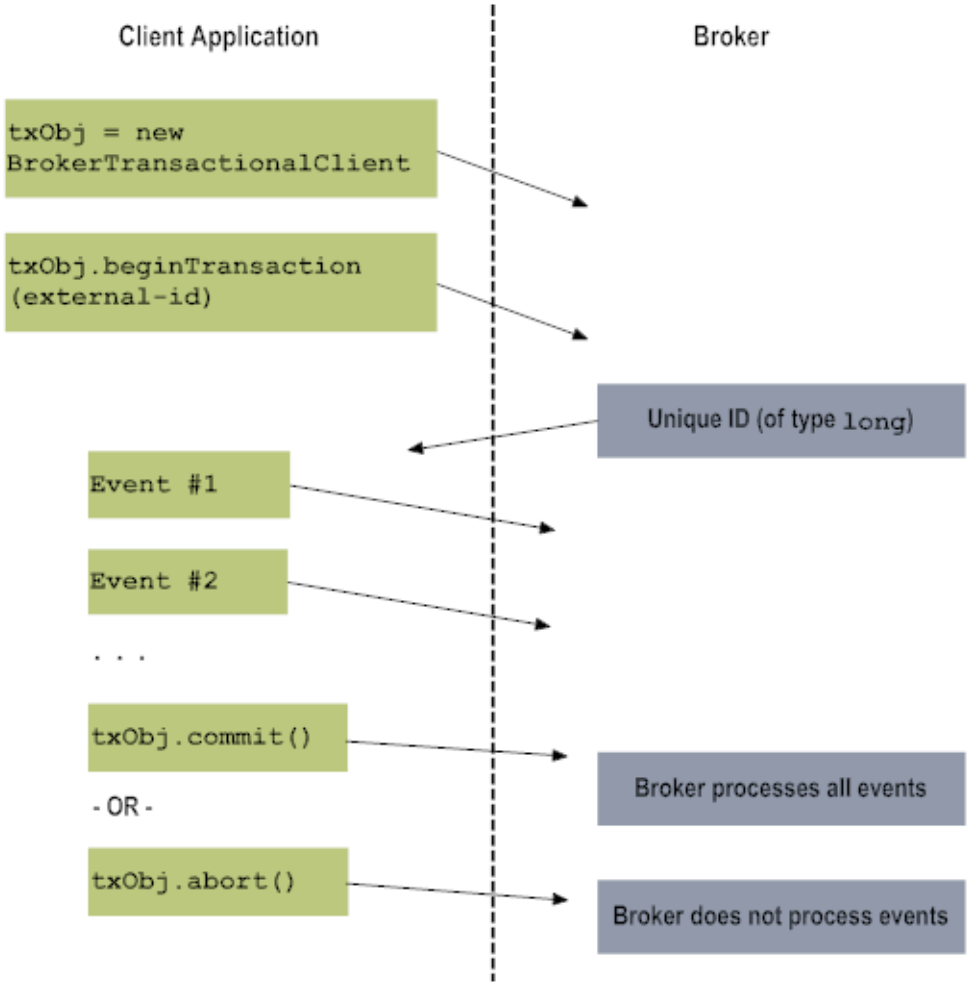
BrokerTransactionalClient class is a subclass of the BrokerClient class. BrokerTransactionalClient is a dedicated Broker Client that is exclusively used for transactional operations. Any task in the transactional operation environment should be done using the BrokerTransactionalClient. All of the existing BrokerClient methods that can be supported through the transactional class are supported by the BrokerTransactionalClient.

Transaction Support

A Broker transactional client initiates a transaction to publish, deliver or get the events that make up the transaction, then ends the transaction by committing or aborting the events. Other features of transaction support are:

- After an `ABORT` operation, all request events in the transaction are discarded and will not generate any reply events, not even error replies.
- All operations throw exceptions.
- Any request event that generates an error on a `COMMIT` operation will cause subsequent events to be discarded. Furthermore, any changes from prior events will be rolled back. If an error occurs, an exception (either BrokerTxClosedException or BrokerInvalidTxException) will be thrown.

Transaction Context Timeline



Note: When the Broker Server is stopped and restarted, all open transactions prior to stopping the Broker Server will be aborted.

Using Broker Transaction Clients

To use Broker transaction clients, your application should follow these steps:

1. Obtain a third-party identifier (such as Tuxedo), which will be used with all publish, deliver, get and acknowledge events that will be part of the transaction. See "Obtaining an External Transaction ID" on page 112 for more information.
2. Begin the transaction. See "Beginning a Transaction" on page 112 for more information.

3. Publish, deliver, acknowledge, or get the events that make up the transaction. See "Operating within a Transaction" on page 112 for more information.
4. End the transaction, which can consist of a `COMMIT` or `ABORT` operation. See "Ending a Transaction" on page 115 for more information.

Obtaining an External Transaction ID

To generate an external transaction ID, use a third-party identifier, such as a Tuxedo. Refer to the documentation of the third-party identifier for more information.

Beginning a Transaction

To begin a transaction, first instantiate a new BrokerTransactionalClient object, then call the transactional client's beginTransaction method.

A unique Broker transaction ID that is specific to this newly opened transaction will be generated and returned as part of the new transaction object. This ID is unique to the specific Broker.

Note that this method can be null or can have an external transaction ID.

```
//unique Broker transaction ID
  long tid;
//Create Transactional Client:
try {
BrokerTransactionalClient pub =
BrokerTransactionalClient.newOrReconnect
(broker_host, broker_name,pubid, client_group_p, "TxTest-Pub", null);
} catch (BrokerException ex)
{
System.out.println ("Failed to create pub client\n" + ex);
System.exit(1);
}
//beginTransaction:
// Start a transaction to publish events
try {
      tid = pub.beginTransaction(null);
} catch (BrokerException ex) {
System.out.println ("Error on starting transaction " +ex);
return;
   }
. . .
```

Operating within a Transaction

The transactional client object obtained at the beginning of the transaction is used to call all the supported transactional publish, deliver, and getEvent(s) methods of the BrokerClient object.

Your application publishes the events that make up the transaction in the manner described in "Publishing and Delivering Events" on page 119.

Note: Any event that is published with a transaction ID that is not known to the Broker will return an exception.

Whether or not your application will receive an acknowledgment event or a reply event for each event it publishes depends on how the event type was defined.

Note: Event types are defined with Software AG Designer. See the *Software AG Designer Online Help* for more information. Note that event types are known as document types in Designer.

Publishing a Transaction

Use the publish method to publish a transaction. This method publishes one event with the given transaction ID. When the event is published, it is sent to the Broker. The Broker then forwards the event to all its subscribing clients.

Note that the event is processed only after the transaction is committed. The following example contains an excerpt that shows the use of the publish method.

```
. . .
// Define the BrokerClient to be used as a transactional client.
   BrokerTransactionalClient pub;

// Define the event to be published
   BrokerEvent e;

// Publish event
   try {
      pub.publish(e);
    } catch (BrokerException ex) {
      System.out.println ("Error on transactional publish" +ex);
      return;
    }
. . .
```

Publishing Multiple Events

Your application can publish several events with one call to the BrokerTransactionalClient.publish method that accepts an array of BrokerEvent objects. The following example contains an excerpt that shows the use of this method, which accepts a BrokerEvent array.

```
. . .
// transactional multi-publish
BrokerEvent[] er;
try {
pub.publish (er);
} catch (BrokerException ex) {
System.out.println ("Error on multi-publish" +ex);
return;
}
. . .
```

Whether or not your application will receive an acknowledgment event or a reply event for each event it publishes depends on how the event type was defined.

Note: Event types are defined with Designer. See the *Software AG Designer Online Help* for more information. Note that event types are known as document types in Designer.

Delivering a Transaction

To deliver a single event in one given transaction, use the deliver method. To deliver multiple events in one given transaction, use the deliver method. These methods send one event or multiple events to the Broker that will forward the events to the Client of the specified Client ID.

These methods require the following parameters:

- *dest_id* . The client identifier of the Broker client that is to receive the events.
- *event(s)* . The BrokerEvent object or the array of BrokerEvent objects to be delivered.

Note that the event is processed only after the transaction is committed.

The following example contains an excerpt that shows the use of the deliver method for delivering a single event in one given transaction.

```
. . .
// Do transacted Deliver
 static String subid = "Sub";
 static String destid = "Sub";

for (int i=0; i<count; i++) {
     try {
       pub.deliver (destid, e);
     } catch (BrokerException ex) {
       System.out.println ("FAIL TEST-A Error on tx deliver" +ex);
       return;
     }
   }
. . .
```

Getting Events

Your application can use the BrokerTransactionalClient.getEvents method to retrieve one or multiple events with a single call, instead of calling BrokerEvent.getEvent to retrieve events one at a time.

Using the BrokerTransactionalClient.getEvents method, the subscribing client is given events only once per session. To get events again before acknowledging, disconnect your subscriber and connect again.

The following example contains an excerpt that shows the use of the BrokerTransactionalClient.getEvents method for retrieving events in one given transaction. This method accepts these parameters:

- *max_events* . The maximum number of events you want to be returned (up to 160).
- *int msecs* . The number of milliseconds to wait for an event, if none are currently available. This can be set to `-1` if you want to block indefinitely.

```
. . .
```

```
// Create subscriber
try {
  sub = BrokerTransactionalClient.newOrReconnect(broker_host,
 broker_name,subid, client_group_s, "TxTest-Sub",
null);
} catch (BrokerException ex){
  System.out.println ("Failed to create pub client\n" + ex);
  System.exit(1);
}

// Loop getEvent and expect to receive events
for (int i=0; i<count; i++){
 try {
  rc_ev = sub.getEvent(5000);
   System.out.println("Received Event\n" + rc_ev.toString());
 }catch (BrokerException ex) {
  System.out.println ("Error on getEvent\n" + ex);
 }
}
. . .
```

Preparing Transactions For Commit

Before you commit the transactions, prepare the transactions for a two-phase commit. Use the prepare method to prepare a transaction, or use the prepareAll method to prepare a given list of transactions. If prepare fails, those transactions can only be aborted. The prepared transactions can be recovered by calling recover. Calling prepare or prepareAll methods is optional. You can directly commit or abort transactions without preparing them.

```
.  .  .
// prepare transaction
try {
      long i = bc.beginTransaction("tx3");
      System.out.println("Transaction id : " +i);
      BrokerEvent ev1 = new BrokerEvent(bc, eventtypename);
      bc.publish(ev1);
      bc.prepare("tx3");
    } catch (BrokerException ex) {
      System.out.println ("Error on prepare" +ex);
      return;
    }
} .  .  .
```

Ending a Transaction

Ending a transaction involves a `COMMIT` or `ABORT` operation.

Committing a Transaction

Use the commit method to commit an open transaction, or use the commitAll method to commit a given list of open transactions.

Note that all work performed on these transactions will be finalized. The following example contains an excerpt that shows the use of the commit method.

```
.  .  .
```

```
// commit transaction
try {
      pub.commit();
    } catch (BrokerException ex) {
      System.out.println ("Error on commit" +ex);
      return;
    }
}
.  .  .
```

Aborting a Transaction

Use the abort method to end an open transaction, or use the abortAll method to end a given list of open transactions. These methods will discard all work previously performed and will invalidate the transaction(s).

The following example contains an excerpt that shows the use of the abort method.

```
.  .  .
// abort transaction
try {
      pub.abort();
    } catch (BrokerException ex) {
      System.out.println ("FAIL abort_publish:Error on transactional abort" +ex);
return;
    }
.  .  .
```

Destroying the Transactional Client

Use the destroy method to destroy a transactional client. When a transactional client is destroyed, its event queue and all other client state information will also be destroyed. The following example contains an excerpt that shows the use of the destroy method.

```
.  .  .
//destroy the transactional client
try {
 pub.destroy();
} catch (BrokerException ex) {
 System.out.println ("Failed to destroy pub client\n" + ex);
 System.exit(1);
}
.  .  .
```

Creating a Transactional Client

To work purely in a transactional environment interacting with transactional methods, only the BrokerTransactionalClient object is required. The following example shows how to create a BrokerTransactionalClient object. Note that the following example also applies to reconnectTransactionalClient() method.

```
public void connect()
{
BrokerTransactionalClient pub, sub;

 /* Create publisher */
 try {
 pub = BrokerTransactionalClient.newOrReconnectTransactionalClient
```

```
(broker_host, broker_name,pubid, client_group_p, "TxTest-Pub",
null);
} catch (BrokerException ex) {
System.out.println ("Failed to create pub client\n" + ex);
System.exit(1);
}

 /* Create subscriber */
 try {
   sub = BrokerTransactionalClient.newOrReconnectTransactionalClient
     (broker_host, broker_name, subid, client_group_s, "TxTest-Sub",
null);

 } catch (BrokerException ex) {
   System.out.println ("Failed to create pub client\n" + ex);
   System.exit(1);
 }
```

8 Publishing and Delivering Events

Publishing Events

When your client application publishes an event, the Broker places it in the event queue of each BrokerClient that has subscribed to that event type.

Note: Event types are defined with Software AG Designer, described in the *Software AG Designer Online Help*. Please note that *events* are known as *document types* in Designer.

In order for your application to be able to *publish* an event, it must first create a BrokerClient, as described on "Creating and Destroying Broker Clients" on page 26. Your application can then use the BrokerClient to create an event, as described on "Creating Events" on page 58. After setting the event fields, as described on "Setting Sequence Fields" on page 64, your application is ready to publish the event.

Limits on Publication

The *client group* to which the BrokerClient belongs defines the event types the BrokerClient can publish. See "Client Groups" on page 25 for more information. The following example contains an excerpt from a sample application that shows the use of the BrokerClient.canPublish method to check for event publication permission. This method simply requires an event type name.

```
. . .
BrokerClient c;
boolean can_publish;
. . .
 /* Check publish permission */
try {
can_publish = c.canPublish("Sample::SimpleEvent");
} catch (BrokerException ex) {
System.out.println("Error on check for can publish\n"+ex);
return;
}
if (can_publish == false) {
System.out.println("Cannot publish event");
System.out.println("Sample::SimpleEvent.");
System.out.println("Make sure it is loaded in the broker and");
System.out.println("permission is given to publish it in the");
System.out.println(client_group + " client group.");
return;
}
. . .
```

You can also use the BrokerClient.getCanPublishNames method to obtain the names of all the event types which a Broker client can publish.

Publishing an Event

After initializing the necessary event fields, your application can publish an event by calling the BrokerClient.publish method. The following example contains an excerpt from a

sample application that shows the use of the BrokerClient.publishEvent method. This method accepts a BrokerEvent.

```
. . .
BrokerClient c;
BrokerEvent e;
. . .
try {
c.publish(e);
} catch (BrokerException ex) {
System.out.println("Error on publish\n"+ex);
return;
}
. . .
```

Publishing Multiple Events

Your application can publish several events with one call to the BrokerClient.publish method that accepts an array of BrokerEvent objects. The following example contains an excerpt that shows the use of this method, which accepts a BrokerEvent array.

```
. . .
BrokerClient c;
BrokerEvent e[5];
. . .
/* Create and initialize the event array */
. . .
try {
c.publish(e);
} catch (BrokerException ex) {
System.out.println("Error on publish\n"+ex);
return;
}
. . .
```

The BrokerClient.publishWithAck method can be used by your Broker client to publish one or more events while, at the same time, acknowledging the receipt of one or more events. For more information on event acknowledgement, see "Using Sequence Numbers" on page 133.

Delivering Events

In addition to publishing events to potentially many Broker clients, the webMethods Broker system also allows your application to *deliver* an event to a single Broker client, through the Broker. In order to deliver an event to a specific Broker client, your application must have the client identifier of that client.

Note: The recipient of a delivered event does not have to register a subscription for the event type being delivered. The recipient only needs to be permitted to receive the delivered event type, as specified by the client group of the receiving Broker client.

Obtaining the Client Identifier

You can either hard code the client identifier of the recipient, if it is well known, or you can extract the identifier from a event that you have received from the Broker client as shown in the following example.

```
. . .
BrokerEvent e;
String client_id;
. . .
try {
client_id = e.getStringField("_env.pubId");
} catch (BrokerException ex) {
System.out.println("Error on getting destination ID\n"+ex);
return;
 }
 . . .
```

Delivering an Event

The following example shows the use of the BrokerClient.deliver method to deliver a single event. This method accepts these parameters:

- A string containing the destination identifier (client identifier of the recipient).
- The BrokerEvent to be delivered.

Note: Delivering an event to a Broker client that no longer exists will not return an error.

```
. . .
BrokerClient c;
BrokerEvent e, event;
String dest_id;
. . .
/* create a client */
. . .
/* open any subscriptions */
. . .
/* receive an event */
. . .
/* create the event to be sent and set the event fields */
. . .
/* obtain the client id of the recipient from the received event */
try {
dest_id = event.getStringField("_env.pubId");
} catch (BrokerException ex) {
System.out.println("Error on getting destination ID\n"+ex);
return;
}
/* Deliver the event to the recipient */
try {
c.deliver( dest_id, e);
} catch (BrokerException ex) {
System.out.println("Error on delivering event\n"+ex);
return;
}
. . .
```

Delivering Multiple Events

The following example shows the use of the BrokerClient.deliver method to deliver multiple events. This method accepts these parameters:

- A string containing the destination identifier (client identifier of the recipient).
- The array of events to be delivered.

Delivering multiple events

```
. . .
BrokerClient c;
BrokerEvent e[5];
BrokerEvent event;
String dest_id;
...
/* create the events to be sent and set their event fields */
. . .
/* obtain the client id of the recipient */
try {
dest_id = event.getStringField("_env.pubId");
} catch (BrokerException ex) {
System.out.println("Error on getting destination ID\n"+ex);
return;
 }
/* Deliver the event to the recipient */
try {
c.deliver( dest_id, e);
} catch (BrokerException ex) {
System.out.println("Error on delivering events\n"+ex);
return;
}
. . .
```

The BrokerClient.deliverWithAck method can be used by your Broker client to deliver one or more events while, at the same time, acknowledging the receipt of one or more events. For more information on event acknowledgement, see "Using Sequence Numbers" on page 133.

9 Handling Errors

Overview

This chapter describes how to handle exceptions thrown by webMethods Broker API methods. Reading this chapter will help you understand:

- How to catch exceptions.
- How to obtain error codes.
- How to set the webMethods Broker system diagnostic level.
- How to display the descriptive text associated with an exception.

BrokerExceptions

Many of the webMethods Broker API methods are designed to throw a BrokerException when an error occurs. The BrokerException class is the base class for all the exceptions that can be thrown by the webMethods Broker API. For a list of all of the possible exceptions, see "API Exceptions" on page 193.

Catching Exceptions

The example below shows how to catch an exception thrown by an webMethods Broker API method invocation. If an exception is thrown, this example code will print out the contents of the exception and return.

```
. . .
BrokerClient c;
. . .
/* Create a client */
try {
c = new BrokerClient(broker_host, broker_name, null,
client_group, "Publish Sample #1",null);
} catch (BrokerException ex) {
System.out.println("Error on create client\n"+ex);
return;
}
. . .
```

Determining the Exception Type

You have two options for determining the exact type of exception that has been thrown.

1. You can develop your code to catch the specific types of exceptions that you expect, as shown in the following example:

```
. . .
BrokerClient c;
. . .
/* Create a client */
try {
```

```
c = new BrokerClient(broker_host, broker_name, null,
client_group, "Publish Sample #1",null);
} catch (BrokerClientExistsException ex) {
System.out.println("Can't create client, already exists\n");
return;
} catch (BrokerNotRunningException ex) {
System.out.println("Can't create client, Broker not   running\n");
 return;
. . .
```

2. You can catch the BrokerException super-class and use the `instanceof` operator to determine which type of exception you have caught, as shown in the following example:

```
. . .
BrokerClient c;
. . .
/* Create a client */
try {
c = new BrokerClient(broker_host, broker_name, null,
client_group, "Publish Sample #1",null);
} catch (BrokerException ex) {
if(ex instanceof BrokerClientExistsException) {
System.out.println("Can't create client, already exists\n");
return;
} else if(ex instanceof BrokerNotRunningException) {
System.out.println("Can't create client, Broker not running\n");
 return;
 . . .
}
. . .
```

Getting an Exception Description

You can use the BrokerException.toString or BrokerException.toCompleteString methods to obtain a descriptive message from BrokerException. The BrokerException.toString method returns a brief message; the BrokerException.toCompleteString method returns a more detailed description.

Setting the System Diagnostic Level

The BrokerException class provides two static methods you can use to query and set the level of error reporting that the webMethods Broker system will present as standard output for your application.

Use the BrokerException.setDiagnostics method to set the level or error reporting.

Use the BrokerException.getDiagnostics method to query the current diagnostic level.

Each of these methods uses the following diagnostic levels:

Diagnostic Level	Description
0	No error output will be produced. Use this setting for deployed applications.

Diagnostic Level	Description
1	Produces error output for major errors only. This is the default setting
2	Produces all error output for all public methods. Use this setting when debugging an application.

10 Using BrokerDate Objects

Overview

This chapter describes the webMethods Broker API for representing and manipulating date and time representations. Reading this chapter should help you understand how to create, set, and query a BrokerDate object.

Date and Time Representation

The webMethods Broker API uses the BrokerDate class to provide a simplified date and time representation for your applications. The BrokerDate class contains the following data members:

Data Member	Description
`int year`	Year value. Must be less than 4096.
`int month`	Month value.
`int day`	Day value.
`int hour`	Hour value.
`int min`	Minute value.
`int sec`	Second value.
`int msec`	Millisecond value.
`boolean is_date_and_time`	Set to `true` if both date and time are represented. Set to `false` if only a date is represented.

BrokerDate Methods

When you construct a new BrokerDate object and specify a date and time, the `is_date_and_time` member is set to true. If you construct an empty BrokerDate, the `is_date_and_time` member is set to false by default.

Other BrokerDate methods include:

Method	Description
`clear`	Clears the entire contents of this BrokerDate.
`clearTime`	Clears only the hour, minute, second, and millisecond.
`compareTo`	Determines if one BrokerDate greater than, equal to, or less than another BrokerDate.
`equals`	Determines if one BrokerDate is equal to another.
`getJavaCalendar`	Converts a BrokerDate to a Java Calendar object.
`getJavaDate`	Converts a BrokerDate to a Java Date object.
`parseDate`	Creates a BrokerDate and sets it to the date and time contained in the specified String.
`parseLocalizedDate`	Creates a BrokerDate and sets it to the date and time contained in the specified String.
`setDate`	Sets the year, month, and day. Year must be < 4096.
`setDateTime`	Sets the date and time.
`setJavaCalendar`	Sets a BrokerDate from a Java Calendar object.
`setJavaDate`	Sets a BrokerDate from a Java Date object.
`setTime`	Set only the hour, minute, second, and millisecond.
`toLocalizedString`	Returns a string representation of a BrokerDate, using a specified Locale.
`toString`	Returns a string representation of a BrokerDate using the `java.util.Locale.US`.

11 Using Sequence Numbers

Overview

This chapter describes how to use event sequence numbers to ensure reliable event delivery. Reading this chapter should help you understand:

- How to use publish sequence numbers.
- How to use receipt sequence numbers.

Sequence Numbers

Sequence numbers in the webMethods Broker system are designed to ensure that events are delivered reliably. The use of sequence numbers is optional, but is recommended if your application requires reliable delivery of events and is capable of storing and tracking these numbers.

Sequence numbers can be used between a publishing application and the Broker or between the Broker and a receiving application. Here are some common characteristics of both publish and receipt sequence numbers:

- Sequence numbers are 64-bit unsigned integers that contain non-zero values.
- Sequence numbers are always incremented and are assumed to never wrap back to 1.

Publisher Sequence Numbers

You do not need to use publisher sequence numbers if your only concern is whether or not a published or delivered event gets to the Broker. If your application's publish or deliver method returns successfully, the event was transmitted to the Broker. If the publish or deliver method fails, you can simply send the event again.

When you re-send an event, there is a small chance a duplicate event will be presented to the Broker. The Broker might successfully receive a published event, but then encounter some error that prevents it from notifying your application. Avoiding this situation can be important if your application is dealing with financial transactions, for example.

When your application uses publisher sequence numbers, it allows the Broker to recognize and discard duplicate events. When used properly, this eliminates the window of opportunity for the presentation of duplicate events to the Broker.

Note: Starting with version 6.1, the use of publisher-assigned sequence numbers has been deprecated for use with transactional clients (BrokerTransactionalClient). However, their use with non-transactional clients (BrokerClient) will continue to be supported.

Using Publisher Sequence Numbers

When the Broker receives an event with a sequence number that has not been set or is set to zero, it assumes that event sequence numbering rules are not to be applied to the event. If your application does not want to use publish sequence numbers, it simply should not set them or set them to zero.

Your application can set a non-zero publish sequence number for an event by calling theBrokerEvent.setPublishSequenceNumber prior to publishing or delivering the event. The BrokerClient.deliverAckReplyEvent and BrokerClient.deliverNullReplyEvent methods allow you to specify a publish sequence number directly.

When the event is published or delivered, the Broker will discard the event if the sequence number is less than or equal to the highest sequence number received from that Broker client so far.

Note: To ensure that events are not discarded, publishing applications should use increasing sequence numbers. The sequence number values, however, can be incremented by a value greater than one.

Maintaining Publish Sequence Number State

To make the most of publisher sequence numbers, publishing applications should store, in some reliable way, the sequence numbers of successfully transmitted events. This will allow them to continue publishing where they left off if they should terminate unexpectedly and need to be restarted.

Your application can use the BrokerClient.getLastPublishSequenceNumber method to determine the highest sequence number that it has used so far.

Your application can obtain the current setting on an event's sequence number by calling the BrokerEvent.getPublishSequenceNumber method.

Receipt Sequence Numbers

Receipt sequence numbers can be used to prevent the Broker from presenting duplicate events to your application's BrokerClient when it retrieves the next event. The Broker always sets sequence numbers on the events it presents to receiving clients. If your receiving client acknowledges an event's sequence number, the Broker will not present that event again.

Note: Volatile events are not assigned sequence numbers by the Broker, so they cannot be specifically acknowledged. Volatile events are deleted from the client's event queue as soon as they are retrieved by a Broker client.

Default Acknowledgment

The following methods will automatically acknowledge all previously retrieved events for your receiving Broker client, effectively ignoring sequence numbers:

- BrokerClient.getEvent
- BrokerClient.getEvents

This default behavior avoids the loss of events from the client's queue in cases where your application crashes while processing an event, because the event is not acknowledged until the next time your Broker client requests an event. The next time your Broker client requests an event, the Broker will assume the last retrieved event has already been processed.

Your Broker client can acknowledge the received event by either asking for another event, using the same get-event method, or by explicitly calling BrokerClient.acknowledge. A Broker client that is implicitly acknowledging events would normally call BrokerClient.acknowledge only if it is planning to exit and wants to acknowledge all previously retrieved events without actually receiving any more events.

Note: Before exiting, Broker clients with an *explicit-destroy* life cycle that are using BrokerClient.getEvent or BrokerClient.getEvents with implicit acknowledgment should explicitly acknowledge the receipt sequence number of the last event received using either BrokerClient.acknowledge or BrokerClient.acknowledgeThrough. Failure to do so will result in the last event being received again the next time you connect the Broker client.

Default Acknowledgment With Callbacks

Your Broker client has much less flexibility in acknowledging events when it registers callback objects to process events, as described in "Using the Callback Model" on page 47. All callback event handling methods must either return true or false. If true is returned, the event will automatically be acknowledged. If false is returned, the event is assumed to have not been successfully handled and, therefore, it is not acknowledged.

Important: If an event meets the criteria for more than one callback object and any one of the callback event handling methods returns false, that event will not be acknowledged.

Explicitly Acknowledging Events

By explicitly acknowledging events, your application retains a greater degree of control over what events the Broker will maintain.

Your Broker client can use the BrokerClient.getEvents method to obtain one or more events from the Broker while simultaneously acknowledging events through a specified

sequence number. If the sequence number is set to -1, no acknowledgment is sent and you can then use one of the following two methods to explicitly acknowledge the events you choose.

Your Broker client can use the BrokerClient.acknowledge method to acknowledge the receipt of a single event with the specified sequence number. If a sequence number of zero is specified, all previously unacknowledged events will be acknowledged.

Your Broker client can use the BrokerClient.acknowledgeThrough method to acknowledge the receipt of all events up to and including the event with the sequence number specified. If a sequence number of zero is specified, all previously unacknowledged events will be acknowledged.

The BrokerClient.deliverWithAck method can be used by your Broker client to deliver one or more events while, at the same time, acknowledging the receipt of one or more events.

The BrokerClient.publishWithAck method can be used by your Broker client to publish one or more events while, at the same time, acknowledging the receipt of one or more events.

Maintaining Receipt Sequence Number State

To make the most of explicitly acknowledging receipt sequence numbers, your receiving applications should store, in some reliable way, the receipt sequence numbers of successfully processed events. This will allow your applications to continue receiving events where they left off if they should terminate and need to be restarted.

You can use the BrokerEvent.getReceiptSequenceNumber method to obtain the receipt sequence number from a received event.

Other Considerations

The Broker guarantees that events from a single publishing client cannot be processed out of order. This has important implications when more than one Broker client is sharing the same event queue because the Broker will not return an event to Broker client which is incapable of acknowledging the event.

The table below shows an example client event queue containing event received from three different publishing clients; Client A, Client B, and Client C.

Publishing Client	Event Queue Position
Client A	1
Client B	2
Client A	3
Client C	4

Publishing Client	Event Queue Position
Client B	5
Client C	6

Consider these steps:

1. Broker client X receives the event from queue position 1 without acknowledging the event.
2. Broker client Y receives the event from queue position 2 without acknowledging the event.
3. Broker client Y then asks for another event and is given the event from queue position 4.

Because the last event from publishing Broker client A has not yet been acknowledged, the next event in the queue from Broker client A cannot be given to a different receiving Broker client.

By enforcing these acknowledgment rules, the Broker allows you to write client applications that process events from a single queue across multiple threads of control.

12 Managing Event Types

Overview

This chapter describes the webMethods Broker objects and methods for managing event types. Reading this chapter will help you understand:

- The information contained in an event type definition.
- How to obtain event type information from a Broker or from a received event.
- How to determine which Broker is managing an event type and the territory to which that Broker belongs.
- How event type definitions are cached and how to manage the cache.
- The information contained in an infoset.
- How to access infosets.

Event Type Definitions

The BrokerTypeDef class contains important information about a particular event type, including the event's fields and their data type. Both the Broker and the webMethods Broker API use event type definitions to verify that an event's fields are set with the correct data types. Event type definitions are also used if you call the BrokerEvent.validate method to validate an event that your application might have received or created.

Several methods are provided to enable your Broker client to obtain event type definitions. After you have obtained an event type definition, you can obtain information on all of the event's fields and their associated data types. Your Broker client might need to obtain event type information to be able to dynamically handle event types whose format has changed since the client application was compiled.

Obtaining Event Type Names

You can obtain an event type definition from an event or using an event type name.

- Use the BrokerClient.getEventTypeNames method to obtain the names of all the event types managed by the Broker to which your Broker client is connected.
- Use the BrokerTypeDef.getTypeName method to obtain the fully qualified event type name from an event type definition.
- Use the BrokerClient.getScopeNames method to obtain the names of all the event types within a particular scope.
- Use the BrokerTypeDef.getBaseTypeName method to obtain the unqualified event type name from an event type definition.

Use the BrokerTypeDef.getScopeTypeName method to obtain the scope name from an event type definition.

Obtaining Event Type Definitions

- Use the BrokerClient.getEventTypeDef method to obtain a single event type definition, given an event type name.
- Use the BrokerClient.getEventTypeDefs method to obtain multiple event type definitions using a list of event type names. You can obtain a list of event type names by using the BrokerClient.getEventTypeNames method.
- Use the BrokerEvent.getTypeDef method to obtain the a single event type definition for an event.
- Use the BrokerClient.getCanPublishTypeDefs method to obtain all the event type definitions which a particular Broker client has permission to publish.
- Use the BrokerClient.getCanSubscribeTypeDefs method to obtain all the event type definitions to which a particular Broker client has permission to subscribe.

Obtaining Event Field Information

Your client application can use an event type definition to obtain information about the event's fields and their associated types.

- Use the BrokerTypeDef.getFieldNames method to obtain a list of all the field names from a event type definition.
- Use the BrokerTypeDef.getFieldType method to obtain the type of a specific event field from a event type definition.
- Use the BrokerTypeDef.getFieldDef method to obtain the definition of a specific event field.

Obtaining Event Type Descriptions

Use the BrokerTypeDef.getDescription method to obtain an event type definition's description.

Obtaining Storage Type Property

An event type's storage type property specifies the how events of its type will be stored by the Broker.

Storage Type	Description
`STORAGE_GUARANTEED`	A two-phase commit process is used to store incoming events on the Broker. This offers the lowest

Storage Type	Description
	performance, but very little risk of losing events if a hardware, software, or network failure occurs
STORAGE_VOLATILE	Local memory is used to store incoming events on the Broker. This offers higher performance along with a greater risk of losing events if a hardware, software, or network failure occurs.

Use the BrokerTypeDef.getStorageType method to obtain the storage type for an event type.

Client groups have two storage modes; *guaranteed* and *volatile*. A client group's storage mode represents the highest storage mode allowed for any events being put into the event queue of a client belonging to that group. With one exception, the Broker will put events into a client's queue using the lesser of the client group's storage mode and the event type's storage mode, as shown in the table below.

If the Client Group Storage Mode is...	And the Event Type Storage Mode is...	Broker will put events into a client's queue using...
Volatile	Volatile	Volatile
Volatile	Guaranteed	Volatile
Guaranteed	Volatile	Volatile
Guaranteed	Guaranteed	Guaranteed

Obtaining the Time-to-live Property

An event type's time-to-live property specifies how long an event will available after being published before it expires and is deleted. Use the BrokerTypeDef.getTimeToLive method to obtain an event type's time-to-live. A time-to-live value of zero means events of that type will never expire.

Obtaining Type Definitions as Strings

You can use the BrokerTypeDef.toString method to obtain a string representation of an event type definition.

Obtaining Broker Information

Each Broker to which a Broker client can connect manages its own set of event type definitions. It can be useful for your client application to obtain information about the Broker that is managing a particular event type definition.

- Use the BrokerTypeDef.getBrokerHost method to obtain host name where the Broker managing a particular event type definition is executing.
- Use the BrokerTypeDef.getBrokerName method to obtain the name of the Broker that is managing a particular event type definition.
- Use the BrokerTypeDef.getBrokerPort method to obtain IP port number of the Broker that is managing a particular event type definition.

Broker Territory and Broker-to-Broker Communication

webMethods Broker allows two or more Broker Servers to share information about their event type definitions and client groups. This enables communication between Broker clients connected to different Brokers. In order to share information, Brokers join a *territory*. All Brokers within the same territory have knowledge of one another's event type definitions and client groups. For more information on this feature, see *Administering webMethods Broker*.

Use the BrokerTypeDef.getTerritoryName method to obtain the territory name of the Broker that is managing this event type definition.

Event Type Definition Cache

The webMethods Broker API keeps a copy of each event type definition used by your application in an *event type cache*. The cache improves the performance of the event field type checking process because the Broker managing the event type does not have to be contacted each time the event type definition is accessed.

The operation of the event type definition cache is usually transparent to your application. Problems can arise if Software AG Designer is used to alter a Broker's event definitions while your application is executing. In these cases, the locally cached event type definitions will no longer be synchronized with the Broker's type definitions. You can use the static methods offered by the BrokerTypeCache class to control the local event type definition cache.

Notification of Event Type Definition Changes

If your application needs to know when a Broker's event type definitions have changed, it can subscribe to the event type `Adapter::refresh`. Whenever this event type is received, your application will know the local event type definition cache needs to be synchronized with the Broker.

Use the BrokerTypeCache.flushCache method to cause the cache contents to be refreshed. This is an advanced method and its use is not required in most situations.

Locking the Type Definition Cache

Multi-threaded applications can run into problems if one thread is accessing event type definition information and another thread calls the BrokerTypeCache.flushCache method. In these situations, threads that need to access the type definition cache should lock the cache before using any of the following methods:

- BrokerClient.getEventTypeDef
- BrokerClient.getEventTypeNames
- BrokerTypeDef.getBrokerHost
- BrokerTypeDef.getBrokerName
- BrokerTypeDef.getBrokerPort
- BrokerTypeDef.getTypeName
- BrokerTypeDef.getFieldNames
- BrokerTypeDef.getFieldType

Use the BrokerTypeCache.lockCache method to lock the local type definition cache and prevent it from being altered or flushed.

After a thread has finished accessing event type definition information, it should unlock the cache using the BrokerTypeCache.unlockCache method.

Infosets

Each event type has a set of one or more infosets that describe configuration information for the event type.

An event type's *public* infoset defines the semantics of the event. In particular, the public infoset defines the event type of a reply that can be expected for an event.

Note: While infosets are neither events nor event types, the methods for obtaining them return the infoset as a BrokerEvent for convenience.

- Use the BrokerClient.getEventTypeNames method to obtain the names of all the infosets defined for a particular event type name.
- Use the BrokerClient.getEventTypeInfoset method to obtain a single infoset, given an event type name and the infoset name.
- Use the BrokerClient.getEventTypeInfosetNames method to obtain a list of infosets, given an event type name and a list of infoset names.

13 Using Event Filters

Overview

This chapter describes the webMethods Broker event filtering facilities, which allow both the Broker and your Broker clients to quickly determine whether an event's contents meet specified criteria. Reading this chapter will help you understand:

- How to specify a filter string.
- How to use a filter string with an event subscription.
- How to use a BrokerFilter within your client application, to match events with filter criteria.

Filter Strings

A filter string specifies criteria for the contents of an event. When you specify a filter string with an event subscription, the Broker uses the filter string to determine which events match your criteria. The Broker will place in your Broker client's event queue just the events that match the filter string.

Filter strings can do any combination of the following:

- Refer to the content of event fields by using *field identifiers*.
- Compare event field contents against constants or computed values.
- Combine event field comparisons using the boolean operators AND, OR, and NOT.
- Perform arithmetic operations on event fields and constants.
- Contain regular expressions.
- Contain string and arithmetic constants.
- Contain a hint that specifies how events should be processed.

Specifying Filter Strings

Consider an event that contains a person's age and the state in which they live. The first event field might have the name `age` and the second might have the field name `state`. The following filter string would match only those events whose `age` field was greater than 65 and whose state field was equal to "FL".

```
age > 65 and state = "FL"
```

In this example filter string, `age` and `state` represent event fields. This filter also contains an arithmetic constant `65` and a string constant "`FL`". The boolean operator `and` is used to combine the field criterion for `age` and `state`.

Other example filter specifications:

```
debt > salary*0.50
```

```
packaging = "portable" and price > 5000
answer = 'Y' or answer = 'y'
(answer = 'Y') or (answer = 'Y')
```

Filter string can also make use of the filter functions described on "Filter Functions" on page 153.

Locale Considerations

The current locale for your application is defined by your operating system, and allows your application to adapt itself to different languages, character sets, time zones, date formats, and monetary conventions.

You can specify a specific locale to be used with your filter at the time you construct a BrokerFilter object.

If you do not specify a particular locale, your filter string is handled using your current locale setting.

Important: The Broker uses a filter string to determine which events match your criteria and that evaluation occurs using the Broker's locale, not your application's locale. This might cause unexpected or unintended event filtering.

The java.text.Collator class is used when comparing strings to provide locale sensitivity. However, this might vary depending on the Java implementation you are using.

Filter Rules

A filter string must adhere to these rules:

1. Field names can be fully qualified, such as

   ```
   struct_field.seq_field[2]
   ```

2. A character constant is a single character surrounded by single quotes, such as 'A'.
3. A string constant is zero or more characters surrounded by double quotes, such as "account".
4. If a character or string constant contains a single or double quote, precede the quote with a backslash. For example: `"\"`
5. Parentheses can be used to control the order of operator precedence, described in "Operator Precedence" on page 150.
6. A division operation must not result in divide by zero.

Field Names

When referring to a structure or sequence field within a filter string, you can use a fully qualified field name. Consider the event type definition shown in the following example.

```
Sample::StructEvent {
string count;
```

```
int scores[];
struct sample_struct {
BrokerDate sample_date;
int sample_array [];
}
}
```

To refer the third integer in the sequence field `scores`, you would use the field name `scores[2]` in your filter string.

To refer to the field `sample_date` within the struct field `sample_struct`, you would use the field name `sample_struct.sample_date` in your filter string.

To refer to the first integer in the sequence field `sample_array`, you would use the field name `sample_struct.sample_array[0]` in your filter string.

Filter Operators

The following tables contain the various operators that you can use to create filters.

Logical Filter Operators

Operator	Description
`!` `not`	Not
`&&` `and`	And
`||` `or`	Or

Comparison Filter Operators

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
==	Equal to

Operator	Description
!=	Not equal to

Arithmetic Filter Operators

Operator	Description
-	Unary minus
*	Multiplication
/	Division
%	Modulus Division
-	Subtraction
+	Addition

Note: Implicit type conversion occurs when operands in an arithmetic operation have different types. The operands are converted to a larger value before the comparison occurs. Type `char` is considered numeric, but `boolean` is not.

Bitwise Operators

Operator	Description
~	Bitwise compliment
<<	Shift left
>>	Shift right
&	Logical and
|	Logical or
^	Logical exclusive or

String Operators

Operator	Description
+	Concatenation
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
==	Equal to
!=	Not equal to

Operator Precedence

The following table describes the precedence and order of evaluation of all of the filter operators. Operators appearing on the same line have the same precedence.

Operator	Description	Order of evaluation
()	Function	left to right
not	logical not	right to left
!	logical not	
~	bitwise not	
-	minus	
*	multiplication	left to right
/	division	
%	modulus division	

Operator	Description	Order of evaluation
-	subtract	left to right
+	add	
<<	left shift	left to right
>>	right shift	
<	less that	left to right
<=	less that or equal to	
>	greater than	
>=	greater than or equal to	
=	equal	left to right
==	equal	
!=	not equal	
&	bitwise and	left to right
^	bitwise exclusive or	left to right
|	bitwise or	left to right
and	logical and	left to right
&&		
or	logical or	left to right
||		

Using Regular Expressions

Regular expressions allow you to specify a complex pattern that is to be matched with an input string. The regexpMatch filter function accepts a regular expression as an argument.

A regular expression is made up of one or more basic components called *atoms*. An atom is used to match a single character in the input string or it can represent multiple occurrences of one or more characters.

If each atom in a regular expression matches a corresponding element in the input string, the expression is said to match the input string.

Characters	Matches
`.`	Any single character in the input string.
`^`	A null string at the start of the input string.
`$`	A null string at the end of the input string.
`\x`	The character `x`. A special character that you want to match in the input string can be escaped in this manner, such as `\^` or `\$`.
`[`*chars*`]`	Any single character from *chars* . Specifying `[a-z]` will match any input string containing all lowercase characters. Specifying `[0-9a-fA-F]` will match any input string containing hexadecimal digits. If a `]` character appears as the first characters in *chars* , it will be treated literally instead of as a terminator.
`[^`*chars*`]`	Any single character that is not contained in *chars* .
`(`*regexp*`)`	Any input string that matches `regexp`. Parentheses can be used to create complex regular expressions.
`*`	A sequence of 0 or more of the preceding atom.
`+`	A sequence of 1 or more of the preceding atom.
`?`	Either a null string or the preceding atom.
`regexp1|` `regexp2`	Anything that matches either `regexp1` or `regexp2`. Note that the `|` delimiter cannot be preceded by or followed by a blank space.

Using Hints

You can include a hint in you subscription by adding a string with the following format.

```
"{hint: HintName=Value}"
```

The table below shows the supported `HintNames` and their associated values.

HintName and Value	Description
`IncludeDeliver=true`	Causes events that are delivered to a Broker client to match a subscription, along with events that are published.
`LocalOnly=true`	In a multi-Broker environment, this causes a subscription to match only those events that originate from the Broker to which this Broker client is connected. Events originating from a different Broker are excluded from the subscription.
`DeadLetterOnly=true`	Creates a subscription to *deadletters* . Deadletters are events to which the local Broker has no registered subscriptions. For information about creating deadletters, see "Detecting Deadletters" on page 94.

Filter Functions

Several functions are provided by the webMethods Broker API that you can use within filter strings. These functions allow you to perform complex string operations, regular expression matches, and string to numeric conversions on event field values. Some examples of the use of functions within filter strings are shown below.

```
charAt(my_string_field,5) =
 charAt("StringConstant",my_i
 startsWith(toUpperCase(myfie
 toString(my_long_field) = "4
 toLong(my_string_field) = 42
```

charAt

```
char charAt(
  string s,
  int index)
```

s — The name of an event field containing a string.

index — The index of the character within the string field that is to be returned. This index is zero-based, so the first character has an index of 0.

Returns the character with the specified *index* from the event field with the name specified by *s* .

contains

```
boolean contains(
  string s,
  string substr)
```

s The name of an event field containing a string.

substr The string that is being searched for.

Returns `true` if the string *substr* occurs within the event field with the name specified by *s* , otherwise `false` is returned.

date

```
date date(
  int year,
  int month,
  int day)
```

year The year.

month The month.

day The day.

Creates a date value with the specified *day* , *month* , and *year* .

date

```
date date(
  int year,
  int month,
  int day,
  int hour,
  int minute,
  int second,
  int millisecond)
```

year The year.

month The month.

day The day.

hour	The hour.
minute	The minute.
second	The second.
millisecond	The millisecond.

Creates a date value with the specified *millisecond* , *second* , *minute* , *hour* , *day* , *month* , and *year* .

endsWith

```
boolean endsWith(
  string s1,
  string s2)
```

s1	The name of an event field containing a string.
s2	The string being searched for.

Returns `true` if the event field with the name specified by *s* ends with the string *s2* , otherwise `false` is returned.

regexpMatch

```
boolean regexpMatch(
  string s,
  string regexp)
```

s	The name of an event field containing a string.
regexp	A string containing a UNIX regular expression.

Returns `true` if the event field with the name specified by *s* matches the UNIX regular expression *regexp* , otherwise `false` is returned.

Important: This function does not support the X/Open regular expression specification. Unicode characters and strings are also not supported by this function.

startsWith

```
boolean startsWith(
  string s1,
  string s2)
```

s1 The name of an event field containing a string.

s2 The string being searched for.

Returns `true` if the event field with the name specified by *s* begins with the string *s2*, otherwise `false` is returned.

substring

```
string substring(
  string s,
  int index1,
  int index2)
```

s The event field from which the substring is to be extracted.

index1 The index of the character within the string field where extraction is to begin. This index is zero-based, so the first character has an index of 0.

index2 The index of the character within the string field where extraction is to end. This index is exclusive.

Returns a substring from the event field with the name specified by *s*, beginning with the character at *index1* and ending at the character at *index2*.

toDouble

```
double toDouble(
  <type> field)
```

field The name of an event field to be converted.

Returns a `double` containing the event field *field*. Any supported field type, except for dates, structures, and sequences, can be converted.

Note: The locale-specific radix character will be recognized when converting a floating point number.

toInt

```
int toInt(
  <type> field)
```

field The name of an event field to be converted.

Returns an `int` containing the event field *field* . Any supported field type, except for dates, structures, and sequences, can be converted.

toLong

```
long toLong(
  <type> field)
```

field The name of an event field to be converted.

Returns a `long` containing the event field *field* . Any supported field type, except for dates, structures, and sequences, can be converted.

toLowerCase

```
string toLowerCase(
  string s)
```

s The name of an event field containing a string.

Returns a copy of the string from the event field with the name specified by *s* , but with all uppercase characters converted to lowercase. Any non-alphabetic or lowercase characters in *s* are returned unaltered.

Note: This function supports locale sensitivity to the extent possible on your particular platform.

toString

```
string toString(
  <type> field)
```

field The name of an event field to be converted.

Returns a string containing the event field *field* , converted to a string. Any supported field type, except for structures and sequences, can be converted.

Note: The locale-specific radix character will be used when converting a floating point type.

toUpperCase

```
char toUpperCase(
  string s,
  int index)
```

s The name of an event field containing a string.

index The index of the character within the string field that is to be returned. This index is zero-based, so the first character has an index of 0.

Returns a copy of the string from the event field with the name specified by *s*, but with all lowercase characters converted to uppercase. Any non-alphabetic or uppercase characters in *s* are returned unaltered.

Note: This function supports locale sensitivity to the extent possible on your particular platform.

toUpperCase

```
char toUpperCase(
  string s)
```

s The name of an event field containing a string.

Returns a single character from the event field with the name specified by *s*, converted to uppercase. If the specified character is non-alphabetic or is already uppercase, it is returned unaltered.

Note: This function supports locale sensitivity to the extent possible on your particular platform.

Using Filters with Subscriptions

When you subscribe to a particular event type using the BrokerClient.newSubscription method, you can specify an optional event filter string. The filter string you specify will be used by the Broker when determining if an event should be placed in your Broker client's event queue. Only those event types for which your Broker client has subscribed that also match the filter specification will actually be placed into your Broker client's event queue. The following example shows how to create an event filter string and use it with an event subscription.

```
BrokerClient c;
String filter_string;
. . .
filter_string = "(A<B) and ((C+12) > (D*3))";
. . .
/* Make a subscription */
try {
c.newSubscription("Sample::SimpleEvent",filter_string);
} catch (BrokerException ex) {
System.out.println("Error on create subscription\n"+ex);
return;
}
. . .
```

Using BrokerFilters

You client application can create a BrokerFilter using the BrokerFilter constructor. A BrokerFilter can be used locally by the client application, in conjunction with the BrokerFilter.match method, to determine whether a particular event type matches the filter.

When creating a BrokerFilter, you must specify an event type name and a filter string. You can then use the BrokerFilter.match method to check any events that your client application might receive.

The following example shows a code excerpt that creates three filters for use by a client application. The BrokerFilter.match method is then used to determine whether the event matches each filter's criteria.

```
. . .
BrokerClient my_client;
BrokerEvent e;
. . .
special = new BrokerFilter(my_client,"Type1",
"(A<B) && ((C+12) >(D*3))");
type1 =new BrokerFilter(my_client, "Type1", null);
type2 = new BrokerFilter(my_client,"Type2", null);

try {
e = c.getEvent(-1);
} catch (BrokerException ex) {
System.out.println("Error on getting event\n"+ex);
return;
}

if (special.match(e)) {
/* A special case of Type1 events that requires special processing */
   . . .
} else if (type1.match(e)) {
/*A Type1 event */
 . . .
} else if (type2.match(e)) {
/* A Type2 event */
. . .
}
. . .
```

To do this same processing without filters, your code would look like that shown in the following example.

```
. . .
BrokerClient my_client;
BrokerEvent e;
String name;
Int a, b, c, d;
. . .
try {
e = c.getEvent(-1);
} catch (BrokerException ex) {
System.out.println("Error on getting event\n"+ex);
return;
}
```

```
name = e.getTypeName();
a = e.getIntegerField("A");
b = e.getIntegerField("B");
c = e.getIntegerField("C");
d = e.getIntegerField("D");
if( (name.equals("Type1") && (a<b) && ((c+12) >(d*3)) ) {
/* A special case of Type1 events that requires special processing */
} else if (name.equals("Type1") {
/*A Type1 event */
} else if (name.equals("Type1") {
/*A Type2 event */
}
. . .
```

Obtaining Filter Strings

You can use the BrokerFilter.getFilterString method to obtain the filter string associated with a Broker filter.

Obtaining Event Type Names

You can use the BrokerFilter.getEventTypeName method to obtain the event type name associated with a Broker filter.

Converting Broker Filters to Strings

The BrokerFilter.toString method allows you to obtain a character string containing the contents of a Broker filter.

14 Load Balancing and Failover for Publish Operations

Overview

This chapter describes the webMethods Broker Java API for implementing client-side load balancing and failover for publish operations. Reading this chapter will help you to understand:

- How to create and destroy a BrokerClusterPublisher.
- How to disconnect or reconnect a BrokerClusterPublisher.
- How to register and use a cluster publisher connection callback.
- How to register and use a cluster publisher selection callback.
- How to include or exclude a specific Broker from the cluster publisher operations.
- How load-balancing is achieved in BrokerClusterPublisher based solution.

Understanding Clustering

When a publish/deliver operation is executed on a valid Broker client connection, it is targeted for the specific Broker to which it is connected. If that Broker is unavailable, the operation fails.

When setting up client-side load balancing and failover, one or more Brokers that share some commonality (in terms of shared metadata and point-to-point connectivity) are brought together by sharing a territory.

In a territory where the metadata is common, publishers and subscribers can be on any of the Brokers in the territory and work as though they are on the same Broker. BrokerClusterPublisher pools together Broker client connections to individual Brokers in a territory and executes publish/deliver operations on any available Broker client connection.

To execute cluster operations, a BrokerClusterPublisher employs one or more Broker client connections to Brokers in a given territory. For an operation to succeed on a Broker client in the cluster pool, the Broker to which it is connected must be part of the same territory during execution. If a Broker becomes unavailable or leaves the territory, it is removed from the cluster pool and is not used until it becomes available again.

BrokerClusterPublisher also provides a means for choosing a specific Broker client connection for executing an operation against the default selection based on a simple *round robin* algorithm. BrokerClusterPublisher provides means for monitoring the connection activity. For example, a Broker leaving or joining the territory or a Broker becoming unavailable or is stopped. It also provides a flexible way of excluding or including a specific Broker on the territory for cluster operations at any point of time during the operations. It also provides some basic capabilities to retrieve permission and statistical information.

BrokerClusterPublishers

BrokerClusterPublisher is a multi-threaded client side solution exposed via Broker Java API applicable only in a territory environment. A BrokerClusterPublisher solution creates two types of client connection; one for monitoring the territory change activity and the other for executing publish/deliver operations as described by the application properties.

Monitoring Territory Activity

BrokerClusterPublisher always maintains a Broker client connection belonging to the system-defined client group "eventLog" for monitoring purposes. This client is used to discover Brokers in a territory at connection time and to detect territory changes such as Brokers leaving or joining that territory. There can be only one such monitoring client for a given BrokerClusterPublisher solution in the territory at any point of time. This is also referred to as *Cluster Monitor Client*. Cluster Monitor subscribes to territory change activity events and acts on them when a change occurs. For example, when a Broker leaves the territory and a Broker client connection to that Broker in the Broker cluster publisher pool exists, the cluster monitor will initiate an operation to remove the Broker client connection to that Broker from the pool. Similarly when a Broker joins the territory, the cluster monitor will initiate an operation to create and add a Broker client connection to the newly joining Broker in to the Broker cluster publisher pool.

Executing Publish/Deliver Operations

The second type of client connection represents actual application Broker client connections. It is based on the application client group and connection parameters as specified at the creation time. For a territory with one or more Brokers, a BrokerClusterPublisher would create a Broker client connection to each of the Brokers on the territory. These are the clients that actually execute publish or deliver operations invoked on the BrokerClusterPublisher object. These Broker client connections are maintained in a cluster connection pool from which the will choose a client to execute an operation. These clients are referred to as Broker*cluster publishers*.

Load Balancing

Load refers to operations that are executed on a BrokerClusterPublisher object which are typically `publish event(s)`, `deliver event(s)`, `publishRequestAndWait` or `deliverRequestAndWait`.

Publishers and Subscribers

When a number of publish or deliver operations are invoked on a BrokerClusterPublisher object, it executes them on one of the Broker client connections from its Broker cluster publisher pool. In this way, the load is distributed onto more than one Broker in the territory setup.

In the classical territory setup, a published event is propagated to other Brokers for subscribers' consumption no matter where the event was published. With this BrokerClusterPublisher feature, a event can be published to a given Broker for the consumption of subscribers on that local Broker only avoiding any event propagation to other Brokers. This introduces a new type of publish operation referred to as *local publish,* which is available only through BrokerClusterPublisher. This enhances the scope of load balancing as far as the event consumption is concerned. A number of local publish operations on the BrokerClusterPublisher with subscribers on each of the territory Brokers would constitute a true load balanced solution. Any change to the event type definition or client group properties is propagated to all the Brokers on the territory, ensuring the integrity of the metadata.

Even for a single subscriber/publisher publishing path in a territory, Broker has some advantages in a BrokerClusterPublisher based solution. When the Broker on which publish operation is being attempted becomes unavailable, BrokerClusterPublisher would automatically retry the operation on a different Broker from the Broker cluster publisher pool until it succeeds. After the event is published successfully, the territory nature will take care of forwarding the event to right Broker where the subscriber is connected.

Failover

BrokerClusterPublisher does not implement any specific functionality to achieve failover, rather it is an inherent property that is delivered with load balancing. When a publish operation is invoked on a BrokerClusterPublisher object, it is then executed on one of the Broker client connections from the Broker cluster publisher pool. If the Broker stops while executing the publish operation, BrokerClusterPublisher will automatically reissue the publish operation on a different Broker client connection in the Broker cluster publisher pool. BrokerClusterPublisher also reissues the operation when the Broker client used for execution represents a Broker that has left the territory.

Using BrokerClusterPublisher

Creating and Destroying BrokerClusterPublisher

BrokerClusterPublisher class has many conventional similarities with Broker client class. It has a constructor that uses parameters like `Broker host`, `Broker name`, `client group name`, `client identifier` and other application client properties as described by a BrokerConnectionDescriptor object. In addition, it also takes a BrokerConnectionDescrptor object that describes the connection parameters fro creating a cluster monitor client. This must be set properly when the system defined client group eventLog is set with ACL information using SSL or access labels.

Creating a BrokerClusterPublisher establishes a connection between your application and one or more Brokers.

Creating a BrokerClientPublisher

Your application can create a BrokerClientPublisher by calling the BrokerClusterPublisher constructor and specifying these parameters.

- The name of the host where the Broker to which you want to connect is executing.
- The name of the Broker to which you want to connect. You can specify a null value if you want to connect to the default Broker. The default Broker for a particular host is determined by your webMethods Broker administrator.
- A unique client ID that identifies your application Broker clients. This identifier cannot be null.
- The client group for your application Broker clients. This client group defines the event types your BrokerClusterPublisher can publish or retrieve, as well as the life cycle and queue storage type for your application Broker clients. Client groups are defined by the webMethods Broker administrator.
- The name of the application that is creating the BrokerClusterPublisher. This name is used primarily by webMethods Broker administration tools. The application name can be any meaningful string of characters of your choosing.
- A BrokerClusterPublisher to be used for the application Broker clients. If you specify a null value, a default connection descriptor will be created for you.
- A BrokerClusterPublisher class to be used for the cluster monitor client. If you specify a null value, a default connection descriptor will be created fro you.

A Broker client belonging to eventLog client group, known as the Cluster Monitor Client, is created to the Broker as specified by the constructor parameters. BrokerClusterPublisher discovers other Brokers on the territory via this Cluster Monitor Client and establishes a Broker client connection to each of these Brokers.

Note: BrokerConnectionDescriptor properties such as state sharing, connection sharing, automatic reconnect and other features except the Access Labels and SSL related are ignored for BrokerClusterPublisher.

The following example contains an excerpt from a sample application that shows the creation of a new BrokerClusterPublisher object.

```
Import COM.activesw.api.client.*;

class ClusterPublish1
{
    static String broker_host = "localhost";
    static String broker_name = null;
    static String client_group = "default";
    . . .
    public static void main(String args[])
    {
        BrokerClusterPublisher bcp;
        . . .
        /* Create */
        try {
           bcp = new BrokerClusterPublisher(broker_host, broker_name,
```

```
            "ClusterPub", client_group, "Cluster Publish Sample #1",null,
            null);
    } catch (BrokerException ex) {
        System.out.println("Error on create cluster publisher\n"+ex);
        return;
    }
   . . .
```

Client Identifiers

The client identifier is a string that uniquely identifies a Broker client. In the context of BrokerClusterPublisher each of the application Broker client that is created on the territory Brokers will have the same client identifier as specified in the constructor. The Cluster Monitor Client will have a default identifier of `CPM-` included with the application client identifier.

For more details on Broker client's client identifier refer to "Client Identifiers" on page 28.

Obtaining Client Identifiers

The best way to obtain an identifier of another Broker client is by retrieving the `pubId` envelope field from an event published by that client, as described in "Obtaining Client Identifiers" on page 28. But you can always use BrokerClusterPublisher.getClientId for obtaining application client identifier and BrokerClusterPublisher.getMonitorClientId for cluster monitor's client ID.

Note: A client identifier cannot start with a # character, nor can it contain `/` or `@` characters. See "Parameter Naming Rules" on page 199 for complete details.

Destroying a BrokerClusterPublisher

The following example contains an excerpt from a sample application that shows the use of destroy method.

```
BrokerClusterPublisher bcp;
. . .
/* Destroy */
try {
       bcp.destroy();
} catch (B  rokerException ex) {
       System.out.println("Error on destroy\n"+ex);
       return;
}
. . .
```

Disconnecting and Reconnecting BrokerClusterPublisher

The webMethods Broker API allows you to disconnect a BrokerClusterPublisher without destroying the underlying Broker client objects. This is only useful if your application Broker client's life cycle is explicit-destroy as the client state for the Broker clients and all

the queued events is preserved by the Broker. When the BrokerClusterPublisher reconnects to the Broker, specifying the same client ID, it can continue processing on the same client state. Disconnecting and reconnecting can be particularly useful for applications that want to do batch-style processing at scheduled intervals.

Disconnecting a BrokerClusterPublisher

You can disconnect your BrokerClusterPublisher by calling the BrokerClusterPublisher.disconnect method as shown in the following example. If the application Broker client's life-cycle is destroy-on-disconnect the client state and the event queue storage will be destroyed by the Broker. If it is explicit-destroy, the client state and event queue storage will be preserved until the Broker clients reconnect.

```
. . .
BrokerClusterPublisher bcp;
. . .
/* Disconnect */
try {
      bcp.disconnect();
} catch (BrokerException ex) {
      System.out.println("Error on disconnect\n"+ex);
      return;
}
. . .
```

Reconnecting a BrokerClusterPublisher

You can reconnect to a previously disconnected BrokerClusterPublisher that has a life cycle of explicit-destroy on the application clients by calling BrokerClusterPublisher.reconnect methods specifying the same client ID that was used when the BrokerClusterPublisher was originally created as shown in the following example.

```
class ClusterPublish1
{
  static String broker_host = "localhost";
  static String broker_name = null;
  static String client_group = "default";
  . . .
  public static void main(String args[])
  {
      BrokerClusterPublisher bcp;
      . . .
      /* Create */
      try {
         bcp = new BrokerClusterPublisher(broker_host, broker_name,
         "ClusterPub", client_group, "Cluster Publish Sample
         #1",null, null);
      } catch (BrokerException ex) {
             System.out.println("Error on create cluster
             publisher\n"+ex);
             return;
}
      . . .
      /* Disconnect */
      try {
             bcp.disconnect();
      } catch (BrokerException ex) {
```

```
            System.out.println("Error on disconnect\n"+ex);
            return;
      }

      /* Reconnect */
      try {
                  bcp = BrokerClusterPublisher.reconnect(broker_host,
      broker_name, "ClusterPub", client_group, "Cluster
      Publish Sample #1",null, null);
      } catch (BrokerException ex) {
            System.out.println("Error on reconnecting cluster
            publisher\n"+ex);
            return;
      }
    . . .
```

Using the newOrReconnect method

You might find it convenient to use the BrokerClusterPublisher.reconnect method to create or reconnect a client when your client application is expected to be executed repeatedly. The newOrReconnect method will attempt to create a BrokerClusterPublisher. If the Broker cluster publishers already exist and was simply disconnected, it will be reconnected.

```
class ClusterPublish1
{
    static String broker_host = "localhost";
    static String broker_name = null;
    static String client_group = "default";
    . . .
    public static void main(String args[])
    {
    BrokerClusterPublisher bcp;
    . . .
    /* Create */
    try {
           bcp = BrokerClusterPublisher.newOrReconnect(broker_host,
    broker_name, "ClusterPub", client_group, "Cluster
    Publish Sample #1",null, null);
    } catch (BrokerException ex) {
    System.out.println("Error on newOrReconnect cluster publisher\n"+ex);
    return;
    }
. . .
```

Broker Cluster Publisher Connection Notification

The connection notification allows you to register a callback method for a BrokerClusterPublisher. The connection notification is invoked when a Broker client is added or removed from the Broker cluster publisher pool as triggered by the territory change activity on cluster monitor client. The connection notification feature is particularly useful for determining which Broker is available for cluster operations when there is a lot of join or leave activity on the territory.

Note: The connection callback method has a global scope and only a single callback can be active at any given time. Registering a callback cancels the

previously registered callbacks. Note that this connection callback on a BrokerClusterPublisher is not the same as the connection callback on a Broker client.

Defining a Connection Callback Object

Use the BrokerCPConnectionCallback interface to derive your own callback object. Your implementation must provide an implementation of the handleConnectionChange callback method. This method indicates the connection change activity within the Broker cluster publisher pool and can be used only for informational purposes apart from logging of the client pool changes.

Registering the Connection Callback Object

Use the BrokerClusterPublisher.registerConnectionCallback method to register a method that you want to be called in the event a Broker client is added or removed from the Broker cluster publisher pool. This method accepts two parameters. The first parameter is the BrokerCPConnectionCallback, it is derived object that implements your callback method. The second parameter is a client_data object, which is used to pass any needed data to the callback method.

Note: Any callback objects previously registered for a client will be replaced by the one currently being registered.

When the BrokerCPConnectionCallback object is the callback method that is invoked, that method's `connect_state` parameter is set to one of the following BrokerClient defined values shown in the table below. Also, the `conn_name` is set to the fully-qualified Broker name on which the change occurred.

connect_state	Meaning
`CONNECT_STATE_DISCONNECTED`	The Broker as specified by `conn_name` has left the territory or down
`CONNECT_STATE_CONNECTED`	The Broker as specified by `conn_name` has joined the territory or became available

Canceling the Connection Callback Object

You can un-register a callback by invoking the BrokerClusterPublisher.cancelConnectionCallback.

```
static class ClusterConnCallback implements BrokerCPConnectionCallback
{
    public void handleConnectionChange(BrokerClusterPublisher bcp,
                                       int state,
                                      String conn_str,
                                      Object client_data) {
```

```
        System.out.print("Connection change on : "+conn_str+ " - ");
        if (state == BrokerClient.CONNECT_STATE_CONNECTED)
            System.out.println(" CONNECTED");
        else if (state == BrokerClient.CONNECT_STATE_DISCONNECTED)
             System.out.println(" DISCONNECTED");
  }
}
public static void main(String args[])
{
  BrokerClusterPublisher bcp;
  ClusterConnCallback conn_cb;
  BrokerEvent e;
  . . .
  /* create BrokerClusterPublisher */
  . . .
  /* create and register connection callback */
  conn_cb = new ClusterConnCallback();
  try {
   bcp.registerConnectionCallback( conn_cb, null)
   } catch (BrokerException ex) {
  System.out.println("Error in registering connection callback");
  return;
  }
  . . .
  /* operations on BrokerClusterPublisher */
  try {
  bcp.cancelConnectionCallback()
  } catch (BrokerException ex) {
  System.out.println("Error in canceling connection callback");
  return;
  }
  /* disconnect or destroy up BrokerClusterPublisher */
  . . .

}
```

Broker Cluster Publisher Selection Notification

The selection notification allows you to register a callback method for a BrokerClusterPublisher that will be invoked whenever a Broker client needs to be chosen from the Broker cluster publisher pool for executing the operation. In the absence of any registered selection callback, BrokerClusterPublisher employs a round robin algorithm to choose Broker client from the Broker cluster publisher pool. If you want to enforce a certain order or an algorithm for choosing a Broker client, you can register a selection callback object that will be invoked just before choosing a Broker client for a publish/deliver operation.

Note: Selection callback method has a global scope and only a single callback can be active at any given time. Registering a callback will cancel the previously registered callbacks if any exists.

Defining a Selection Callback Object

Use the BrokerCPSelectionCallbackinterface to derive your own callback object. Your implementation must provide an implementation of the chooseClusterClient method. This method passes-in the list of available Brokers in the form of an array of BrokerInfo objects-a list of Brokers to which a valid Broker client exists in the Broker cluster publisher pool along with the event(s) to be published/delivered. Your algorithm can examine the list of Brokers and the event(s) and choose an index from the BrokerInfo array that represents the Broker client connection you want to use for the current operation. With this index, BrokerClusterPublisher uses the specified Broker client from the Broker cluster publisher pool and executes the operation on that client connection.

Registering the Selection Callback Object

Use the BrokerClusterPublisher.registerSelectionCallback method to register a method to be called for selecting a Broker client from the Broker cluster publisher pool. You must implement both callback methods on this selection callback object; the first one intended for any publish/deliver operation involving a single event and the second one involving operation on multiple events.

This method accepts two parameters. The first parameter is the BrokerCPConnectionCallback, it is a derived object that implements your callback methods. The second parameter is a client_data object, which is used to pass any needed data to the callback method.

Note: Any callback objects previously registered for a BrokerClient will be replaced by the one currently being registered.

Canceling the Selection Callback Object

You can un-register a callback by invoking the BrokerClusterPublisher.cancelConnectionCallback.

```
static class ClusterSelCallback implements BrokerCPSelectionCallback
{
    int sindex = 0;
    int mindex = 0;

/* implements a round-robin algorithm on the list of available broker clients */
  public int chooseClusterClient(BrokerClusterPublisher bcp,
                                 BrokerEvent[] event,
                                 BrokerInfo[] bi,
                                 Object client_data) {
        System.out.println("#of Broker cluster publishers: "+bi.length);
        for (int i=0;i<bi.length;i++)
            System.out.print(bi[i].broker_name+" ");

        mindex = (mindex+1)%bi.length;
        System.out.println(" ["+bi[mindex].broker_name+"]");
        return mindex;
}

/* implements a round-robin algorithm on the list of available broker clients */
```

```
    public int chooseClusterClient(BrokerClusterPublisher bcp,
                                   BrokerEvent event,
                                   BrokerInfo[] bi,
                                   Object client_data) {
        System.out.println("#of Broker cluster publishers: "+bi.length);
        for (int i=0;i<bi.length;i++)
            System.out.print(bi[i].broker_name+" ");

        sindex = (sindex+1)%bi.length;
        System.out.println(" ["+bi[sindex].broker_name+"]");
        return sindex;
    }
}
public static void main(String args[])
{
    BrokerClusterPublisher bcp;
    ClusterSelCallback sel_cb;
    BrokerEvent e;
    . . .
    /* create BrokerClusterPublisher */
    . . .
    /* create and register connection callback */
    sel_cb = new ClusterSelCallback();
    try {
    bcp.registerSelectionCallback( sel_cb, null)
    } catch (BrokerException ex) {
    System.out.println("Error in registering selection callback");
    return;
    }
    . . .
    /* operations on BrokerClusterPublisher */
    try {
    bcp.cancelSelectionCallback()
    } catch (BrokerException ex) {
    System.out.println("Error in canceling selection callback");
    return;
    }
    /* disconnect or destroy up BrokerClusterPublisher */
    . . .

}
```

Obtaining BrokerClusterPublisher Status

There are a variety of methods you can use to obtain state, status, and statistical information about a BrokerClusterPublisher object.

- Use the BrokerClusterPublisher .getApplicationName method to obtain the name of the application associated with this BrokerClusterPublisher.
- Use the BrokerClusterPublisher.getClientGroup method to obtain the name of the client group with BrokerClusterPublisher is associated.
- Use the BrokerClusterPublisher.getClientId method to obtain the BrokerClusterPublisher client ID.
- Use the BrokerClusterPublisher.getMonitorClientId method to obtain the BrokerClusterPublisher Cluster Monitor's client ID.

- Use the BrokerClusterPublisher.getTerritoryName method to obtain the name of the territory.
- Use the BrokerClusterPublisher.canPublish method to determine whether a BrokerClusterPublisher can publish an event of a specific event type.
- Use the BrokerClusterPublisher.getCanPublishNames method to get a list of all event types that can be published by the BrokerClusterPublisher.
- Use the BrokerClusterPublisher.getClusterPublisherInfo method to obtain a string that contains information on Broker clients on the Broker cluster publisher pool and other cluster information of the BrokerClusterPublisher in the form of an event.
- Use the BrokerClusterPublisher.getClusterPublisherStats method to obtain the statistics on the number of events published, delivered and received along with other useful information of the BrokerClusterPublisher.
- Use the BrokerClusterPublisher.toString method to obtain a string that contains information on Broker clients on the Broker cluster publisher pool and other cluster information of the BrokerClusterPublisher.
- Use the BrokerClusterPublisher.isConnected method to determine whether a BrokerClusterPublisher is currently connected to a Broker.

Publishing and Delivering Events

A simple publish or deliver operation invoked on the BrokerClusterPublisher object is executed on one of the valid Broker client connections from the Broker cluster publisher pool. If an error occurs during the execution, the operation will be either retried on a different Broker client on the Broker cluster publisher pool or returned as failed with proper exception. A failed operation on a Broker client from the Broker cluster publisher pool will not be retried on the same Broker client. BrokerClusterPublisher will retry on each of the Broker client connections from the cluster pool until the operation succeeds on one of the connections or returns an error when all of the Broker client connections are exhausted.

Creating a New BrokerEvent

Before your BrokerClusterPublisher application can publish or deliver an event it must create the event and set the event's fields. The following example contains an excerpt from a sample application that shows the creation of a new event. The constructor takes the following parameters:

- A BrokerClusterPublisher reference
- The name of the event type. In the following example, the event scope is `Sample` and the event type name is `SimpleEvent`.

```
class ClusterPublish1
{
    static String broker_host = "localhost";
    static String broker_name = null;
```

```
    static String client_group = "default";
    . . .
    public static void main(String args[])
    {
        BrokerClusterPublisher bcp;
        BrokerEvent e;
        . . .
        /* Create a BrokerClusterPublisher */
        . . .
        /* Create a BrokerEvent */
        try {
        e = new BrokerEvent( bcp, "Sample::SimpleEvent");
        } catch (BrokerException ex) {
        System.out.println("Error on creating new BrokerEvent \n"+ex);
        return;
        }
. . .
```

Field Type Checking

When you create an event with a BrokerClusterPublisher reference, the following field type checking rules are applied to the event.

1. All event fields will be set on the event at the time the event is created.
2. You cannot set a field that does not exist for the event type.
3. You cannot set an event field with a data type other than that defined by the event type.

When you create an event with a null BrokerClusterPublisher reference, the following type checking rules are applied to the event.

1. You can set fields with any field name and any data type.
2. Any attempt to retrieve an event field that was not previously set will cause a BrokerFieldNotFoundException exception to be thrown.
3. After a field has been set, you are not allowed to change the field's type without first clearing the event field, using the BrokerEvent.clearField method or clearing the entire event using the BrokerEvent.clear method.

Publishing Events

When your client application publishes an event, the Broker places it in the event queue of each BrokerClient that has subscribed to that event type. In order for your application to be able to publish an event, it must first create a BrokerClusterPublisher. Your application can then use the BrokerClusterPublisher to create an event, as described in the previous example. After setting the event fields, your application is ready to publish the event.

The client group to which the BrokerClusterPublisher belongs defines the event types the BrokerClusterPublisher can publish. See "Client Groups" on page 25 for more information.

The following example contains an excerpt from a sample application that shows the use of the BrokerClusterPublish.canPublish method to check for event publication permission. This method requires an event type name.

```
. . .
BrokerClusterPublisher bcp;
boolean can_publish;
. . .
/* Check publish permission */
try {
       can_publish = bcp.canPublish("Sample::SimpleEvent");
} catch (BrokerException ex) {
       System.out.println("Error on check for can publish\n"+ex);
       return;
}
if (can_publish == false) {
       System.out.println("Cannot publish event");
       System.out.println("Sample::SimpleEvent.");
       System.out.println("Make sure it is loaded in the broker and");
       System.out.println("permission is given to publish it in the");
       System.out.println(client_group + " client group.");
       return;
}
. . .
```

You can also use the BrokerClusterPublisher.getCanPublishNames method to obtain the names of all the event types which a BrokerClusterPublisher can publish.

Publishing an Event

After initializing the necessary event fields, your application can publish an event by calling the BrokerClusterPublisher.publish method. The following example contains an excerpt from a sample application that shows the use of the BrokerClusterPublisher.publish method. This method accepts a BrokerEvent.

BrokerClusterPublisher can publish an event that is targeted for subscribers on the Broker on which the event is published or to all the subscribers on any Broker in the territory.

The following example illustrates how to publish a single event to territory subscribers:

```
. . .
BrokerClusterPublisher bcp;
BrokerEvent e;
. . .
try {
       bcp.publish(e);
} catch (BrokerException ex) {
       System.out.println("Error on publish\n"+ex);
       return;
}
. . .
```

The following example illustrates how to publish a single event to subscribers on the local Broker only:

```
. . .
BrokerClusterPublisher bcp;
BrokerEvent e;
. . .
```

```
try {
       bcp.localPublish(e);
} catch (BrokerException ex) {
       System.out.println("Error on publish\n"+ex);
       return;
}
. . .
```

Publishing Multiple Events

Your application can publish several events with one call to the BrokerClusterPublisher.publish method that accepts an array of BrokerEvent objects. The following example contains an excerpt that shows the use of this method, which accepts a BrokerEvent array.

The following example illustrates how to publish multiple events to territory subscribers:

```
. . .
BrokerClusterPublisher bcp;
BrokerEvent e[5];
. . .
/* Create and initialize the event array */
. . .
try {
       bcp.publish(e);
} catch (BrokerException ex) {
       System.out.println("Error on publish\n"+ex);
       return;
}
. . .
```

The following example illustrates how to publish multiple events to subscribers on the local Broker only:

```
. . .
BrokerClusterPublisher bcp;
BrokerEvent e[5];
. . .
/* Create and initialize the event array */
. . .
try {
       bcp.localPublish(e);
} catch (BrokerException ex) {
       System.out.println("Error on publish\n"+ex);
       return;
}
. . .
```

Delivering Events

In addition to publishing events to potentially many Broker clients, the webMethods Broker system also allows your application to deliver an event to a single Broker client, through the Broker. In order to deliver an event to a specific Broker client, your application must have the client identifier of that client.

Note: The recipient of a delivered event does not have to register a subscription for the event type being delivered. The recipient only needs to be permitted to receive the delivered event type, as specified by the client group of the receiving Broker client.

You must hard code the client identifier of the recipient, if it is well known.

Delivering an Event

The following example shows the use of the BrokerClusterPublisher.deliver method to deliver a single event. This method accepts these parameters:

- A string containing the destination identifier (client identifier of the recipient).
- The BrokerEvent to be delivered.

The following example illustrates how to deliver a single event:

```
. . .
BrokerClusterPublisher bcp;
BrokerEvent e, event;
String dest_id = "DestClient";
. . .
/* create a BrokerClusterPublisher */
. . .
/* create the event to be sent and set the event fields */
. . .
/* Deliver the event to the recipient */
try {
       bcp.deliver( dest_id, e);
} catch (BrokerException ex) {
       System.out.println("Error on delivering event\n"+ex);
       return;
}
. . .
```

Delivering Multiple Events

The following example shows the use of the BrokerClusterPublisher.deliver method to deliver a single event. This method accepts these parameters:

- A string containing the destination identifier (client identifier of the recipient).
- An array of BrokerEvents to be delivered.

The following example illustrates how to deliver multiple events:

```
. . .
BrokerClusterPublisher bcp;
BrokerEvent e, event;
String dest_id = "DestClient";
. . .
/* create a BrokerClusterPublisher */
. . .
/* create an array of events to be sent and set the event fields */
. . .
/* Deliver events to the recipient */
```

```
try {
       bcp.deliver( dest_id, e);
} catch (BrokerException ex) {
       System.out.println("Error on delivering events\n"+ex);
       return;
}
. . .
```

Request-Reply Model

You can use the request-reply model for applications that publish or deliver a request event to a server application which is expected to return a reply event. The reply event can contain data or can simply be an acknowledgment with no data.

A tag envelope field is set by the BrokerClusterPublisher internally to identify the request event that it is sending. When a server application receives the request event and prepares the reply event, it ensures that the same tag field is set for the reply as was received on the request event. If your requestor sends several different request events, it should set each with a different tag field. When your requestor receives a reply event, it can check the tag field to determine the original request with which the reply is associated. BrokerEvent.getTag and BrokerEvent.setTag methods are used to get and set tag field values from an event.

Using publishRequestAndWait

BrokerClusterPublisher solution does not employ callback mechanism to achieve synchronous request-reply results. Rather it uses publish followed by getEvents approach to implement request-reply mechanism. In this way the automatic failover on the publish operation is available for request-reply model as well.

The following example illustrates how to use BrokerClusterPublisher.publishRequestAndWait for subscribers on the local Broker:

```
public static void main(String args[])
{
   BrokerClusterPublisher bcp;
   BrokerEvent e, events[];
   int i;
   . . .
   /* create BrokerClusterPublisher */
   . . .
   /* create the request event to be sent and set the event fields */
   . . .
   /* publish the request and wait for a reply */
   try {
   events[] = bcp.localPublishRequestAndWait(e, 6000);
   } catch (BrokerException ex) {
   . . .
   }
. . .
```

The following example illustrates how to use BrokerClusterPublisher().publishRequestAndWait for subscribers in a territory:

```
public static void main(String args[])
{
```

```
   BrokerClusterPublisher bcp;
   BrokerEvent e, events[];
   int i;
   . . .
   /* create BrokerClusterPublisher */
   . . .
   /* create the request event to be sent and set the event fields */
   . . .
   /* publish the request and wait for a reply */
   try {
   events[] = bcp.publishRequestAndWait(e, 6000);
   } catch (BrokerException ex) {
   . . .
   }
. . .
```

Include-Exclude Brokers

At the creation time, the cluster monitor discovers the Brokers on the territory and BrokerClusterPublisher creates BrokerClient connections to these Brokers with the specified application settings. A territory can contain a mixture of Brokers from different versions of webMethods Broker. BrokerClusterPublisher can utilize Brokers from version webMethods Broker 6.0 or higher. All other Brokers are automatically excluded from the cluster operations; therefore, BrokerClusterPublisher automatically excludes pre-6.0 version Brokers at creation time.

You can also exclude or include a specific Broker from cluster operations by invoking BrokerClusterPublisher.excludeBroker method and BrokerClusterPublisher.includeBroker method respectively. If you discover that a set of Brokers have less of a load or the Broker host has more resources than others, you might want to reroute all operations to a specific set of Brokers on the territory.

The following example illustrates how to use BrokerClusterPublisher().exclude and include methods:

```
public static void main(String args[])
{
   BrokerClusterPublisher bcp;
   BrokerEvent e, events[];
   int i;
   . . .
   /* create BrokerClusterPublisher */
   . . .
   /* create the request event to be sent and set the event fields */
   . . .
   /* When you find that the broker b1 is overloaded and you want to exclude
  the broker from the cluster operations */
   try {
    bcp.excludeBroker(b1@localhost);
   } catch (BrokerException ex) {
   . . .
   }
   . . .
   /* When you find that the broker b1 has returned to normal and you want to
   include the broker for future cluster operations */
   try {
    bcp.includeBroker(b1@localhost);
   } catch (BrokerException ex) {
```

```
   . . .
    }
. . .
```

When a Broker is excluded from on a BrokerClusterPublisher, the Broker client connection to that Broker from the Broker cluster publisher pool is removed immediately so that the Broker client connection can no longer be used. Similarly, after you include the Broker for cluster operations, a new Broker client connection is created so that Broker and future operations will utilize this connection.

15 Working with Basic Authentication

Overview

This chapter explains how to configure basic authentication support for webMethods Broker clients by writing to the Broker Java Client API.

Basic Authentication

webMethods Broker uses the basic authentication mechanism to allow a client program to provide credentials in the form of a user name and password when making a request to access Broker Server. For more information about basic authentication, please see *Administering webMethods Broker*.

Configuring the Broker Java Client for Basic Authentication

This section describes how to use the Broker Java client API COM.activesw.api.client.BrokerConnectionDescriptor class to configure and manage basic authentication. This class contains the methods to get and to set basic authentication related settings. For Java APIs, all basic authentication information is passed by the BrokerConnectionDescriptor object. When a BrokerConnectionDescriptor object is created, the default basic authentication settings will be set by reading the corresponding Java property values.

Configuring Basic Authentication by using BrokerConnectionDescriptor Class

For your Java client, use the BrokerConnectionDescriptor.setAuthInfo method described on page to enable basic authentication, prior to creating a BrokerClient.

When a BrokerConnectionDescriptor is created, the authentication information will be set to null by default. Therefore, you must use the setAuthInfo method before creating a Broker client if you want to enable basic authentication.

Return Type	Method Signature and Description

void	`public void setAuthInfo(`
	`  String username,`
	`  String password)`
	
	Where *username* is the user name for basic authentication and *password* is the password for the basic authentication user.

String	`getAuthUserName()`

The following code sample shows how to configure the settings for Broker basic authentication by using the BrokerConnectionDescriptor.setAuthInfo method, passing in the parameters described in the table above:

```
// Build the connection descriptor.
BrokerConnectionDescriptor connectionDescriptor =
new BrokerConnectionDescriptor()
String username  = "sag";
String password = "SoftwareAG123";

// Create a new Broker client.
client = new BrokerClient(broker_host, broker_name, client_id,client_group,
app_name, connectionDescriptor)
```

Configuring Basic Authentication by Using System Properties

The following table helps you enable basic authentication by using the system properties on Java clients.

To set this basic authentication property...	Set this system property...
User name	BROKER_CONNECTION_BASIC_AUTH_USER As shown here: `System.setProperty("BROKER_CONNECTION_ BASIC_AUTH_USER","sag");`
Password	BROKER_CONNECTION_BASIC_AUTH_PASSWORD As shown here: `System.setProperty("BROKER_CONNECTION_ BASIC_AUTH_PASSWORD "," SoftwareAG123");`

16 Working with SSL

Overview

This chapter explains how to configure SSL support for webMethods Broker clients by writing to the Broker Java Client API. Information is provided that explains how to:

- Authenticate Broker clients.
- Manage SSL certificates for Java client applications.
- Enable encryption.

> **Note:** When implementing SSL using the Broker Java Client API, you will be using JSSE (Java Secure Sockets Extension) for Broker clients version 7.1 and later. The JSSE library is part of the Java 5 JDK package.

Broker SSL Security

The Broker SSL security model provides the following forms of protection for your event-based Broker Java client applications:

- **User authentication.** Authentication verifies the identity of a Broker Server to a Broker Java client attempting to make a connection *and* that of the Broker Java client to the Broker Server (two-way authentication). For a connection to be made, both parties must have first been assigned SSL identities.
- **User authorization** for Broker objects protected by Access Control Lists (ACLs). Only clients whose SSL identities are specified in a Broker object's ACL may connect to that object. This type of security protects confidential data from access by unauthorized users.
- **Encryption** of the data traffic between a Broker client and the Broker Server, to protect sensitive data. This type of encryption is independent of the SSL authentication process and of the ACL authorization process. Typically, you encrypt the data traffic when working with highly sensitive data, or to protect data of a confidential nature that passes across a public network.

Certificate Files

To configure SSL for a Broker Java client, you must first store its SSL client certificate information (public certificate/private key pair) in a keystore file, and the trusted root (public certificate of the client certificate issuer, or CA) in a truststore file. These certificates are used to create the Broker Java client's SSL *identity*.

For the client to make an SSL connection to the Broker Server, you must also have assigned SSL identities to the Broker Server. Once the SSL certificate for the Broker

Server is configured, you can create an SSL connection between the Broker Java client and its Broker Server.

Detailed information about creating and managing keystores and truststores for the Broker Server and the Broker admin component is provided in *Administering webMethods Broker*.

Configuring the Broker Java Client for SSL

This section describes how to use the Broker Java client API `COM.activesw.api.client.BrokerConnectionDescriptor` class to configure and manage SSL. This class contains the methods to get and to set SSL-related settings. For Java APIs, all SSL information is passed by the `BrokerConnectionDescriptor` object. When a `BrokerConnectionDescriptor` object is created, the default SSL settings will be set by reading the corresponding Java property values.

Configuring SSL Using the BrokerConnectionDescriptor Class

For your Java client, use the BrokerConnectionDescriptor.setSSLCertificate method to configure SSL security, prior to creating a BrokerClient.

When a BrokerConnectionDescriptor is created, the certificate file will be set to `null` by default. Therefore, you must use the setSSLCertificate method before creating a Broker client if you want to configure SSL security.

Return Type	Method Signature and Description

`void`	`public void setSSLcertificate(`
	`  string keystore_file,`
	`  string truststore_file,`
	`  KeystoreType keystore_type,`
	`  TruststoreType truststore_type,`
	`  string password)`
	`  throws BrokerException;`
	
	Where:
	■ `keystore_file` is the keystore file path.
	■ `truststore_file` is the truststore file path.
	■ `keystore_type` is the keystore type.
	■ `truststore_type` is the truststore type.
	■ `password` is the password.

`BrokerSSLCertificate`	`getSSLCertificate(`
	`  string keystore_file,`
	`  string truststore_file,`
	`  KeystoreType keystore_type,`
	`  TruststoreType truststore_type,`
	`  string password)`
	`  throws BrokerException`

Return Type	Method Signature and Description
	Where: ■ `keystore_file` is the keystore file path. ■ `truststore_file` is the truststore file path. ■ `keystore_type` is the keystore type. ■ `truststore_type` is the truststore type. ■ `password` is the password. Use this method to get the Entrust SSL certificate.
`TruststoreType`	`getSSLTruststoreType()`
`String`	`getSSLTruststore()`
`KeystoreType`	`getSSLKeystoreType()`
`String`	`getSSLKeystore()`

The following code sample shows how to configure the settings for Broker SSL authentication using the BrokerConnectionDescriptor.setSSLCertificate method, passing in the parameters described in the previous table:

```
// Build the connection descriptor.

  BrokerConnectionDescriptor connectionDescriptor =
                           new BrokerConnectionDescriptor()

  String keystore_file = "C:\mykeystore.p12";
  String truststore_file = "C:\mytruststore.jks";
  KeystoreType keystoreType = "KeystoreType.p12";
  TruststoreType truststoreType = "TruststoreType.jks";
  String password = "1234";


  connectionDescriptor.setSSLCertificate (keystore_file,
                                          truststore_file,
                                          keystoreType,
                                          truststoreType,
                                          password)

  // Set other properties for the connectionDescriptor, if needed.

  // Create a new Broker client.

  client = new BrokerClient (brokerHost, brokerName, clientName,
           clientGroup, ... , "publisher", connectionDescriptor);
```

Configuring SSL by Using the System Properties

You can also use the System Properties to configure SSL. The following table helps you configure SSL by using the system properties on Java clients.

To set this SSL property...	Set this system property...
Keystore	BROKER_CONNECTION_SSL_KEYSTORE As shown here: `System.setProperty("BROKER_CONNECTION_SSL_KEYSTORE", "C:\mykeystore.p12");`
Truststore	BROKER_CONNECTION_SSL_TRUSTSTORE As shown here: `System.setProperty("BROKER_CONNECTION_SSL_TRUSTSTORE", "C:\mytruststore.jks");`
Keystore Type	BROKER_CONNECTION_SSL_KEYSTORE_TYPE As shown here: `System.setProperty("BROKER_CONNECTION_SSL_KEYSTORE_TYPE", "KeystoreType.p12");`
Truststore Type	BROKER_CONNECTION_SSL_TRUSTSTORE_TYPE As shown here: `System.setProperty("BROKER_CONNECTION_SSL_TRUSTSTORE_TYPE", "TruststoreType.jks");`
Ciphersuites	BROKER_CONNECTION_SSL_CIPHERSUITES As shown here: `System.setProperty("BROKER_CONNECTION_SSL_CIPHERSUITES", "TLS_RSA_WITH_AES_128_CBC_SHA");`
Password	BROKER_CONNECTION_SSL_PASSWORD As shown here: `System.setProperty("BROKER_CONNECTION_SSL_PASSWORD", "SoftwareAG123");`
Provider	BROKER_CONNECTION_SSL_PROVIDER As shown here: `System.setProperty("BROKER_CONNECTION_SSL_PROVIDER", "Entrust");`

To set this SSL property...	Set this system property...
DN	BROKER_CONNECTION_SSL_DN As shown here: `System.setProperty("BROKER_CONNECTION_SSL_DN", "cn=brokerclient1, o=webM,st=CA,c=US");`
FIPS	BROKER_CONNECTION_SSL_FIPS As shown here: `System.setProperty("BROKER_CONNECTION_SSL_FIPS", "true");`
TCP No Delay	BROKER_CONNECTION_TCP_NODELAY As shown here: `System.setProperty("BROKER_CONNECTION_TCP_NODELAY", "true");`

Setting Encryption

The BrokerConnectionDescriptor.setSSLEncrypted method allows you to encrypt the data traffic from a Java client to the Broker Server after SSL authentication has taken place.

Important: When a `BrokerConnectionDescriptor` object is created, by default, its encryption flag is set to `true`.

To control whether to use data encryption, invoke the setSSLEncrypted method on the `BrokerConnectionDescriptor` object before creating your Broker client. Use a setting of `true` to enable encryption, and `false` to disable encryption, as shown in the example below (the "on" setting is commented out):

```
/* Configure encryption */

  try {

  connectionDescriptor.setSSLEncrypted( false );  //  Encryption = OFF
// connectionDescriptor.setSSLEncrypted( true );  // Encryption = ON

  catch (BrokerException ex) {
  System.out.println("Error on setSSLEncrypted.\n"+ex);
  return;
  }
```

Retrieving a Broker Server's Certificate

You can use the BrokerClient.getBrokerSSLCertificate method to obtain information about the SSL certificate of the Broker Server to which your Broker client is connected. A getBrokerSSLCertificate object is returned by this method and contains information such as the Broker's distinguished name and the certificate authority that issued the Broker's certificate.

Enabling FIPS in Java Clients

You can enable FIPS mode only if your SSL provider is Entrust. Perform the following steps to enable the FIPS mode in Java clients.

To enable FIPS mode in client

1. Set the SSL provider system property `BROKER_CONNECTION_SSL_PROVIDER`, to `Entrust` as follows:

```
System.setProperty("BROKER_CONNECTION_SSL_PROVIDER", "Entrust");
```

2. To enable the FIPS mode, set the system property, `BROKER_CONNECTION_SSL_FIPS` to `true` as follows.

```
System.setProperty("BROKER_CONNECTION_SSL_FIPS","true");
```

A API Exceptions

This appendix describes the exceptions that can be thrown by the webMethods Broker Java API to report API, communications, and Broker failures.

You can use the BrokerException.toString method to obtain a character string that briefly describes the error associated with a particular BrokerException.

The BrokerException.toCompleteString method lets you obtain a character string that specifically describes the error associated with a particular BrokerException.

BrokerBadStateException
An API call was made that conflicts with the current system state. For example:

- You attempted to register a callback for a subscription ID before registering a general callback.
- You attempted to invoke a second BrokerClient.dispatch while in a callback method.

BrokerClientContentionException
An attempt was made to reconnect a Broker client, but the client is either already in use, or the client has shared state and the maximum number of shared clients has already been reached.

BrokerClientExistsException
The client ID specified when creating a new BrokerClient is already is use.

BrokerCommFailureException
A generic communications fault has occurred. Network failures cause this exception to be thrown.

BrokerConnectionClosedException
The connection to the Broker closed before or during the operation you requested.

BrokerCorruptDataException
The data object on which you are operating is corrupt. Currently only detected on BrokerEvent objects.

BrokerException
This class is the base exception type for all exceptions thrown by Information Broker client classes. It refers to specific exception sub-class for information about why the exception is thrown.

BrokerFailureException
An unexpected failure happened while the Broker was processing your request. This exception can have two possible meanings:

- The API could not correctly process an exception into one of the other exceptions.

- A Broker failure has occurred, such as running out of memory or a corrupted data store.

BrokerFieldNotFoundException
An attempt was made to operate on a BrokerEvent field that does not exist.

BrokerFieldTypeMismatchException
The specified event field is not of the expected type. For example, using the BrokerEvent.setStringField method on an event field of type `int` will cause this exception to be thrown.

BrokerFileNotFoundException
The specified file could not be found or could not be opened.

BrokerFilterParseException
An error occurred while parsing the filter string specified when creating a new BrokerFilter.

BrokerFilterRuntimeException
The filter specified with the BrokerFilter.matchFilter method caused a runtime error, such as division by zero.

BrokerFormatException
This can result from some protocol failures. It is mainly issued by the BrokerString calls to parse values out of strings.

BrokerHostNotFoundException
The host specified when creating or reconnecting a BrokerClient could not be found.

BrokerIncompatibleVersionException
The Broker is running an older version of the product than this API.

BrokerInterruptedException
A BrokerClient.getEvent or BrokerClient.getEvents method was interrupted by the invocation of the BrokerClient.interruptGetEvents method.

This exception can also be thrown if the BrokerClient.dispatch method was interrupted by a call to the BrokerClient.interruptDispatch method.

BrokerInvalidAcknowledgementException
An attempt was made to acknowledge a sequence number that was out of order or which was not been assigned to your Broker client.

BrokerInvalidClassException
The *storage_class* passed to the BrokerEvent constructor could not be instantiated using the class' newInstance method.

BrokerInvalidClientException
The BrokerClient object passed to the method is either disconnected or has been destroyed.

BrokerInvalidClientIdException
The client ID specified when creating a new BrokerClient contained illegal characters or was longer than 255 characters.

BrokerInvalidDescriptorException
The BrokerConnectionDescriptor object passed to the method has been deleted.

BrokerInvalidEventException
The BrokerEvent object passed to the method has been deleted.

BrokerInvalidEventTypeNameException
An event type name contained illegal characters, reserved words, or has components over 255 characters in length. See "Parameter Naming Rules" on page 199 for information on valid event type names.

BrokerInvalidFieldNameException
A field name contains illegal characters, reserved words, or has components over 255 characters in length. See "Parameter Naming Rules" on page 199 for information on valid event type names.

BrokerInvalidPlatformKeyException
An invalid platform key was specified.

BrokerInvalidSubscriptionException
A null *event_type_name* parameter, or a negative subscription ID parameter, or a filter string with a parse error were used when creating a new BrokerSubscription.

This exception can also be thrown if the subscription specified on BrokerClient.cancel was not found by the Broker.

BrokerInvalidTypeCacheException
An internal error occurred with the event type definition cache. This does not represent a user error.

BrokerInvalidTypeDefException
The BrokerTypeDef object passed to the method has been deleted. Either the client it was associated with was disconnected or destroyed, or the event type definition cache was flushed.

BrokerInvalidTypeException
An attempt was made to set an event's field to a value that didn't match the field's type while using a method such as BrokerEvent.setField.

This exception can also be thrown when internal failures occur.

BrokerNoPermissionException
You do not have the necessary permission to do the operation you attempted. For example:

- Attempting to write to a read-only envelope field.
- Using a client group that does not include your identity with the correct permissions when reconnecting or creating a new BrokerClient.
- Attempting to publish an event type that is not allowed by your BrokerClient object's client group.

For more information on client group permissions and can publish permissions see *Administering webMethods Broker*.

BrokerNotImplementedException
The method you requested is not implemented.

BrokerNotRunningException
While attempting to create or reconnect a BrokerClient, the specified host was found but no Broker was running on that host.

BrokerNullParamException
A null value was passed for a parameter that requires a value.

BrokerOutOfRangeException
A parameter value is outside the accepted range. For example, getting a sequence subset using negative indexes will cause this exception to be thrown.

BrokerPublishPauseException
An internal failure occurred while communicating with the Broker.

BrokerProtocolException
An internal failure occurred while communicating with the Broker.

BrokerSecurityException
A security problem occurred that prevented the operation from being completed.

BrokerSubscriptionExistsException
You attempted to create a new BrokerSubscription with an event type and filter that has already been used for another subscription.

BrokerTimeoutException
The requested operation timed out. This occurs for methods like BrokerClient.getEvent when an event is not received within the specified time-out interval.

BrokerTxClosedException
Action attempted on a BrokerClient transaction that has already been either committed or aborted.

BrokerUnknownBrokerNameException
The Broker specified when reconnecting or creating a new BrokerClient was not found.

BrokerUnknownClientGroupException
The client group specified when reconnecting or creating a new BrokerClient was not found.

BrokerUnknownClientIdException
The Client ID specified when reconnecting a BrokerClient was not found.

BrokerUnknownEventTypeException
The specified event type was not found on the Broker. This exception can be thrown, for example, if you attempt to create a new BrokerEvent with a non-existent type.

BrokerUnknownInfosetException
The specified infoset was not found for the event type.

BrokerUnknownKeyException
The platform specified key for BrokerClient.getPlatformInfo has no value defined.

BrokerUnknownNameException
The specified distinguished name does not exist in the appropriate SSL certificate file.

BrokerUnknownTxException
Specified name does not exist in the resource being accessed.

B Parameter Naming Rules

Overview

This appendix describes the rules for naming webMethods Broker and system parameters, including host names, distinguished names, passwords, Broker names, and client group names as well as event type names and event field names.

Note: Unicode character values described in this section are represented as `\u####`.

Length Restriction

webMethods Broker uses a network data representation that requires all 2-byte unicode characters to be converted to 6 byte ANSI strings. Because the maximum parameter length is 255 bytes, this means that a parameter containing only Unicode characters cannot be any longer that 42 bytes. Each of the following webMethods Broker parameters must have a length of either 1 to 255 ANSI characters or 1 to 42 Unicode characters:

- Broker name
- Client group
- Client ID
- Event type name
- Event field name
- Infoset name
- Infoset field name
- Territory name

Restricted Characters

With a few restrictions, an webMethods Broker parameter can be specified using any Unicode or ANSI characters. This allows you to use a variety of languages when naming items such as a Broker or event type. However, some characters are restricted and cannot be used.

- All non-printable ANSI characters (defined as the two ranges `\u0000` to `\u001F` and `\u007F` to `\u009F`).
- The ANSI characters '`@`', '`/`', and '`\`'.

Note: In addition to these restricted characters, specific types of webMethods Broker parameters can place further restrictions on allowable characters. See " webMethods Broker Parameters" on page 202 for complete details.

Reserved Words

`any`	`boolean`	`byte`	`char`
`const`	`date`	`double`	`enum`
`false`	`final`	`float`	`int`
`long`	`null`	`short`	`string`
`struct`	`true`	`typedef`	`unicode_char`
`unicode_string`	`union`	`unsigned`	

EventType and Infoset Names

The name of an event type or infoset cannot be any of the words shown in "Reserved Words" on page 201 above or the table below. The entire event type or infoset name is checked against these two lists of reserved words. This means that you cannot use the name "`broker`" for an event type, but you can use the name "`my::broker`".

`acl`	`broker`	`client`	`clientgroup`
`event`	`eventtype`	`extends`	`host`
`import`	`infoset`	`server`	

System Parameters

The table below shows common restrictions on system parameters, which depend on your specific platform.

System Parameter	Restrictions
Broker Host name	Limited by most systems to printable 7-bit ASCII.
File names and Passwords	Limited by most systems to 8-bit ANSI characters.
Distinguished Names	Limited to printable 7-bit ASCII characters.

webMethods Broker Parameters

The table below shows the restrictions placed on various webMethods Broker parameters.

webMethods Broker Parameter	Restrictions
Territory name Broker name Client Group name Client Id	Cannot begin with a '#" or contain any of the restricted characters described in "Restricted Characters" on page 200.
Application name Platform Info Key	Cannot contain any of the restricted characters described in "Restricted Characters" on page 200.
Event Type name Event Field name Infoset name Infoset Field name	Cannot begin with a digit (0-9) or with an underscore. This can only contain alphanumeric characters, underscores, dollar symbols ('$'), and Unicode characters greater than \u009F. This cannot contain symbols, whitespace, and non-printable ANSI characters.

webMethods Broker Parameter	Restrictions
Platform Info value	No restrictions.
Filter strings	No restrictions, other than the syntax restrictions described in "Using Event Filters" on page 145.
Format strings	No restrictions, but the '$', '{', and '}' characters are used as part of the format syntax.