IBM WebSphere Business Integration Adapters

Connector Development Guide for Java

Note!
Before using this information and the product it supports, read the information in "Notices" on page 521.

20February2004

This edition of this document applies to IBM WebSphere InterChange Server, version 4.2.2, IBM WebSphere Business Integration Adapter Framework, version 2.4, and to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this documentation, email doc-comments@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this document

The IBM(R) WebSphere(R) Business Integration Adapters portfolio supplies integration connectivity for leading e-business technologies and enterprise applications. The system includes tools and templates for customizing, creating, and managing components for business process integration.

This document describes the development of Java connectors in the IBM WebSphere business integration system.

Audience

This document is for connector developers. It assumes proficiency in the Java programming language. The document also assumes a basic familiarity with the IBM WebSphere business integration system, including connectors and business objects.

Related documents

The complete set of documentation describes the features and components common to all WebSphere Business Integration Adapters installations, and includes reference material on specific components.

Note: This document covers the development of connectors written in Java. The development of C++ connectors is documented in the *Connector Development Guide for C++*.

You can install the documentation available for this product or read it directly online at the following sites:

- For general adapter information, for using adapters with WebSphere message brokers (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Business Integration Message Broker), and for using adapters with WebSphere Application Server:

  http://www.ibm.com/websphere/integration/wbiadapters/infocenter
- For using adapters with WebSphere InterChange Server:

  http://www.ibm.com/websphere/integration/wicserver/infocenter
- For more information about message brokers (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Business Integration Message Broker):

  http://www.ibm.com/software/integration/mqfamily/library/manualsa/
- For more information about WebSphere Application Server:

  http://www.ibm.com/software/webserver/appserv/library.html

These sites contain simple directions for downloading, installing, and viewing the documentation.

Typographic conventions

This document uses the following conventions:

`courier font`	Indicates a literal value, such as a command name, file name, information that you type, or information that the system prints on the screen.
italic	Indicates a new term the first time that it appears.
italic, `italic`	Indicates a variable name or a cross-reference.
blue outline	A blue outline, which is visible only when you view the manual online, indicates a cross-reference hyperlink. Click include the outline to jump to the object of the reference.
{ }	In a syntax line, curly braces surround a set of options from which you must choose one and only one.
[ ]	In a syntax line, square brackets surround an optional parameter.
...	In a syntax line, ellipses indicate a repetition of the previous parameter. For example, `option[,...]` means that you can enter multiple, comma-separated options.
< >	In a naming convention, angle brackets surround individual elements of a name to distinguish them from each other, as in `<server_name><`*`connector_name`*`>tmp.log`.
/, \	In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes. All WebSphere Business Integration Adapters product pathnames are relative to the directory where the product is installed on your system.
%text% and *$text*	Text within percent (%) signs indicates the value of the Windows `text` system variable or user variable. The equivalent notation in a UNIX environment is *$text*, indicating the value of the *text* UNIX environment variable.
ProductDir	Represents the directory where the product is installed. For the IBM WebSphere InterChange Server environment, the default product directory is "IBM\WebSphereICS". For the IBM WebSphere Business Integration Adapters environment, the default product directory is "WebSphereAdapters".

Markup conventions

In some chapters, you will find text identified by the following markup:

WebSphere InterChange Server
Describes functionality of the IBM WebSphere business integration system when InterChange Server is the integration broker.

WebSphere MQ Integrator Broker
Describes functionality of the IBM WebSphere business integration system when WebSphere MQ Integrator Broker is the integration broker.

New in this release

This chapter describes the new features of IBM WebSphere business integration system that are covered in this document.

New in WebSphere Business Integration Adapters v2.4.0

February 2004

With this update to the IBM WebSphere Business Integration Adapter 2.4.0 release, the *Connector Development Guide for Java* is provided *only* as part of the IBM WebSphere Business Integration Adapter documentation set. It is no longer provided as part of the IBM WebSphere InterChange Server documentation set.

In addition, information about how to create connector startup scripts has been enhanced. For more information, see "Creating startup scripts" on page 215.

December 2003

For a list of updates and new features for the IBM WebSphere Business Integration Adapter 2.4.0 release, please refer to "New in WebSphere InterChange Server v4.2.2 and WebSphere Business Integration Adapters v2.4.0."

New in WebSphere InterChange Server v4.2.2 and WebSphere Business Integration Adapters v2.4.0

The IBM WebSphere InterChange Server 4.2.2 release and the IBM WebSphere Business Integration Adapter 2.4.0 release provide the following new functionality in the Java connector library:

- A Java connector now uses the IBM Java Object Request Broker (ORB) instead of the third-party VisiBroker ORB.
- A Java connector can now support access to the serialized data sent or or received from a data handler. In previous releases, the connector could access serialized data as a Java String. It can now provide access in any of the following new forms.
  - As an input stream: `boToStream()` and `streamToBO()`
  - As a Java byte array: `boToByteArray()` and `byteArrayToBo()`
  - As a Reader object: `readerToBO()`

  In addition, all data-handler methods now support the ability to identify the character encoding and locale for the data handler to associate with the serialized data. For more information, see the descriptions of these methods in "Calling a data handler" on page 75.
- The Java connector library now provides the following features for an event store (which the `CWConnectorEventStore` class represents):
  - The `setEventStoreStatus()` method has been renamed to `setEventStatus()` to better identify its functionaliy. This method sets the status of an event.
  - The `getBO()` method now provides the ability to return an integer status value to its calling method. The default implementation of `getBO()` continues

to use the form that does *not* provide an internal status value. For more information, see the description of the `getBO()` method in the `CWConnectorEventStore` class.

- The Java Connector Development Kit (JCDK) now provides a more consistent way to create startup scripts for Java connectors. It also provides a template (for both Windows and UNIX-based systems) for the creation of this startup script. For more information, see "Starting up a new connector" on page 213.

In addition, the Adapter Development Kit (ADK) now includes an adapter sample in the `DevelopmentKits\Twineball_sample` subdirectory of the product directory. This adapter sample includes a Java connector.

New in WebSphere InterChange Server v4.2.1 and WebSphere Business Integration Adapters v2.3.0

The IBM WebSphere InterChange Server 4.2.1 release and the IBM WebSphere Business Integration Adapter 2.3.0 release provide the following new functionality in the Java connector library:

- The connector can now provide additional configuration to a data handler when it calls the data handler. The following methods support a *config* argument to specify this additional information:
  - `boToString()`
  - `stringToBo()`

  For more information, see the descriptions of these methods in Chapter 22, "CWConnectorUtil class," on page 347.
- The Java connector library now provides access to individual name-value pairs in application-specific information through new forms of the `getAppText()` method in the `CWConnectorBusObj` class.

  For more information, see the description of this method in Chapter 13, "CWConnectorBusObj class," on page 257.
- In support of duplicate event elimination (which provides guaranteed event delivery), the Java connector library provides the `setDEEId()` method in the `CWConnectorBusObj` class to enable a connector to set a business object's ObjectEventId attribute with the event identifier (ID). For more information, see "Guaranteed event delivery for connectors with non-JMS event stores" on page 136 and the description of the `setDEEId()` method in Chapter 13, "CWConnectorBusObj class," on page 257.
- The Java connector library now provides the ability to modularize the instantiation of an event-store object from its event-store factory with the following features:
  - The getEventStore() method (in the `CWConnectorAgent` class) instantiates an event-store object from its event-store factory. The `CWConnectorAgent` class provides a default implementation of this method. However, you can override it for custom behavior. The default implementation of the `pollForEvents()` method now calls this `getEventStore()` method to obtain its event-store object
  - The `EventStoreFactory` connector configuration property can contain the name of the event-store-factory class for your event store. The `getEventStore()` method (in the `CWConnectorAgent` class) obtains the name of the event-store-factory class it uses from the `EventStoreFactory` property.

  For more information, see "CWConnectorEventStoreFactory interface" on page 178.

- The Java connector library now provides the getTerminate() and setTerminate() methods (in the `CWConnectorEventStore` class) to allow the `pollForEvents()` method to better handle the application-timeout (`APPRESPONSETIMEOUT`) condition.
- The Java connector library now provides verb constants for the Exists and RetrieveByContent verbs. The `VERB_EXISTS` and `VERB_RETRIEVEBYCONTENT` verb constants are defined in the `CWConnectorConstant` class.
- To supplement changes to the return codes of the `gotApplEvents()` method, the manual now provides more information on how to respond to these different outcome-status values. In addition, the `pollForEvents()` method has been enhanced to take these same responses. For more information, see "Sending the business object" on page 189.
- The Java connector library now supports the creation of a custom business object handler through a custom-business-object-handler class, which implements the `CWCustomBOHandler` interface. If your connector supports a business object that requires different processing for one of its verbs, you can create a custom business object handler to handle that verb for the business object. For more information, see "Creating a custom business object handler" on page 174.

New in WebSphere Business Integration Adapters 2.2.0

The IBM WebSphere Business Integration Adapter 2.2.0 release provides the following new functionality in the Java connector library:

- The ″CrossWorlds″ name is no longer used to describe an entire system or to modify the names of components or tools, with are otherwise mostly the same as before. For example ″CrossWorlds System Manager″ is now ″System Manager″ and ″CrossWorlds InterChange Server″ is now ″WebSphere InterChange Server″.
- The Java connector library provides access to hierarchical connector configuration properties with the following enhancements:
  - The `CWProperty` class provides methods that allow you to obtain string values and child properties within a hierarchical connector property. For more information, see Chapter 25, "CWProperty class," on page 391.
  - The `CWConnectorUtil` class provides two new methods to allow you to retrieve the top-level hierarchical connector properties:
    - To retrieve all top-level hierarchical connector properties: `getAllConfigProperties()`
    - To retrieve a specified top-level hierarchical connector property: `getHierarchicalConfigProp()`

  For more information, see "Retrieving hierarchical connector configuration properties" on page 73.

  Note: The Java connector library still provides support for the old single-valued, simple connector property values, though the `getConfigProp()` method.
- The Java connector library now supports duplicate event elimination to provide guaranteed event delivery. Duplicate event elimination is most often used by JMS-enabled adapters that have event stores that are *not* implemented as JMS queues. Use the `DuplicateEventElimination` connector property to enable this functionality. For more information, see "Guaranteed event delivery for connectors with non-JMS event stores" on page 136.
- The Java connector library now provides the following API methods:
  - The getSupportedVerbs() method (in the `CWConnectorBusObj` class) provides a list of the business object's supported verbs.

  - The setLocale() method (in the CWConnectorBusObj class) allows you to set the locale that is associated with a business object. This new method complements the getLocale() method that has already been defined in this same class.
  - The cleanupResources() method (in the CWConnectorEventStore class) allows you to release resources that the event store has used.
- Chapter 8, "Adding a connector to the business integration system," on page 209 now provides more information on how to add a Java connector to the WebSphere business integration system, including:
  - How to create an initial configuration file for a connector
  - How to create a startup script for a Java connector from a sample startup file
  - Use of the new CWConnEnv.bat (Windows) or CWConnEnv.sh (UNIX) file for system-variable settings
- Chapter 2, "Designing a connector," on page 37 now provides more information on how to internationalize a connector.
- Several Java connector library methods have been changed to better handle status return codes:
  - The default implementation of the pollForEvents() method now takes the following actions:
    - It handles the CONNECTOR_NOT_ACTIVE and NO_SUBSCRIPTION_FOUND status return codes from its call to the gotApplEvent() method. For more information, see "Sending the business object" on page 189.
    - It returns an outcome status of APPRESPONSETIMEOUT if access to the event store fails. Failure to access the event store can occur in any of the following event-store methods:

Event-store method	Exception raised
fetchEvents()	StatusChangeFailedException
archiveEvent()	ArchiveFailedException
deleteEvent()	DeleteFailedException
updateEventStatus()	StatusChangeFailedException

  - The agentInit() method now returns an outcome status of FAIL if, when it throws an exception, the exception-detail object's status value is not set. If the status value *is* set within the exception-detail object, agentInit() returns that status value.
  - The doVerbFor() method now returns an outcome status of APPRESPONSETIMEOUT if, when it throws a ConnectionFailureException, the exception-detail object's status value is not set. If the status value *is* set within the exception-detail object, doVerbFor() returns that status value.

New in WebSphere Business Integration Adapters 2.1

The IBM WebSphere Business Integration Adapter 2.1 release provides the following new functionality in the Java connector library:

- The Java connector library provides access to attribute values that are LongText with the following new methods in the CWConnectorBusObj class:
  - getLongTextValue() to retrieve a LongText attribute value
  - setLongTextValue() to set a LongText attribute value

- The Java connector library now supports synchronous sending of an event with the `executeCollaboration()` method in the `CWConnectorAgent` class. This method is valid for use *only* when InterChange Server is the integration broker.

New in WebSphere Business Integration Adapters 2.0.1

The IBM WebSphere Business Integration Adapter 2.0.1 release provides an internationalized version of the Java connector library. This internationalized connector library enables you to develop adapters that can be localized for many different locales (A locale includes culture-specific conventions and a character code set.). The structure of connectors has changed in the following ways to accomodate locales:

- The connector framework now has a locale associated with it. This locale is determined either from the operating system locale or from configuration properties. The Java connector library provides the `getGlobalEncoding()` and `getGlobalLocale()` methods in the `CWConnectorUtil` class to access this information from within the connector.
- A business object has a locale associated with it. This locale is associated with the data in the business object, *not* with the name of the business object definition or its attributes. The Java connector library provides the `getLocale()` method in the `CWConnectorBusObj` class to obtain the name of this locale from within the connector.

For more information, see "An internationalized connector" on page 55.

New in WebSphere Business Integration Adapters 2.0

The IBM WebSphere Business Integration Adapter 2.0 release provides support for adapters. An *adapter* is a set of software modules that communicate with an integration broker and with applications or technologies to perform tasks such as executing application logic and exchanging data. For an introduction to adapters and integration brokers, see "Adapters in the WebSphere business integration system" on page 3.

In addition, the structure of IBM WebSphere business integration system documentation for the development of connectors has changed in this release:

- IBM introduces a new application programming interface (API) for the development of Java connectors. Features of this connector library include:
  - Classes to encapsulate an event and event store within the Java connector: `CWConnectorEvent`, `CWConnectorEventStore`
  - A single class, `CWConnectorBusObj`, to provide access to the business object, business object definition, and attributes
  - Classes to provide more information in exceptions that methods of the Java connector library throw: `CWException`, `CWConnectorExceptionObject`
  - Other classes retain the functionality of the old low-level Java connector library by being wrappers for the old classes.

  IBM recommends this new Java connector library for all new development of Java connectors. For a summary of the classes and methods of this connector library, see Chapter 9, "Overview of the Java connector library," on page 233 Support for the old low-level Java connector library will be continued for backward compatibility.
- The following guides have been combined to create a single document that covers the development of Java connectors:

Connector Development Guide	Material on how to develop a connector is now found in Parts I and II of this new document.
Connector Reference: Java Class Library	Reference material on the low-level Java connector library is now found in Part IV.

Reference material on the new Java connector library is now found in Part III of this document.

Part 1. Getting started

Chapter 1. Introduction to connector development

This chapter provides a brief overview of connectors in the IBM WebSphere business integration system. It also introduces the Java Connector Development Kit (JCDK) and summarizes the development steps you need to follow to implement a connector. This chapter contains the following sections:

Adapters in the WebSphere business integration system

The *IBM WebSphere business integration system* consists of the following components, which allow heterogeneous business applications to exchange data:

- A set of IBM WebSphere Business Integration Adapters

  An IBM WebSphere Business Integration Adapter, called simply an *adapter*, provides the components to support communication between an integration broker and either applications or technologies to perform tasks such as executing application logic and exchanging data.
- An integration broker

  The task of an *integration broker* is to integrate data among heterogeneous applications. The IBM WebSphere business integration system can include either of the integration brokers in Table 1..

Table 1. Integration brokers in the WebSphere business integration system

Integration broker	For more information	Documentation set
WebSphere message brokers (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Business Integration Message Broker)	*Implementing Adapters for WebSphere Message Brokers*	WebSphere Business Integration Adapters
WebSphere Application Server	*Implementing Adapters for WebSphere Application Server*	WebSphere Business Integration Adapters
IBM WebSphere InterChange Server (ICS)	*Implementation Guide for WebSphere InterChange Server*	WebSphere InterChange Server

In the IBM WebSphere business integration system, the integration broker communicates to these applications through adapters. The following adapter components actually provide this communication:

- "Business objects" on page 5, whose role is to hold information about an application event
- "Connectors" on page 6, whose role is to send information about an application event to an integration broker or to receive information about a request from the integration broker.

Figure 1 shows how these components transfer information from an application to an integration broker.

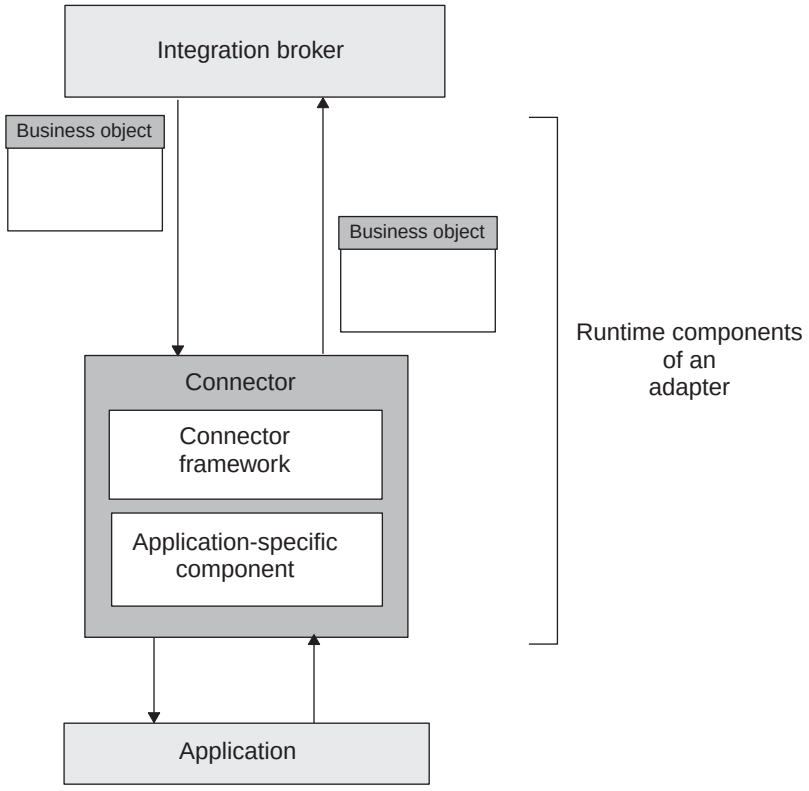

Figure 1. Adapter components that provide information transfer

Note: An adapter also includes configuration and development components. For more information, see "Tools for adapter development" on page 27.

Figure 2 shows the WebSphere business integration system and the role that connectors play within this system.

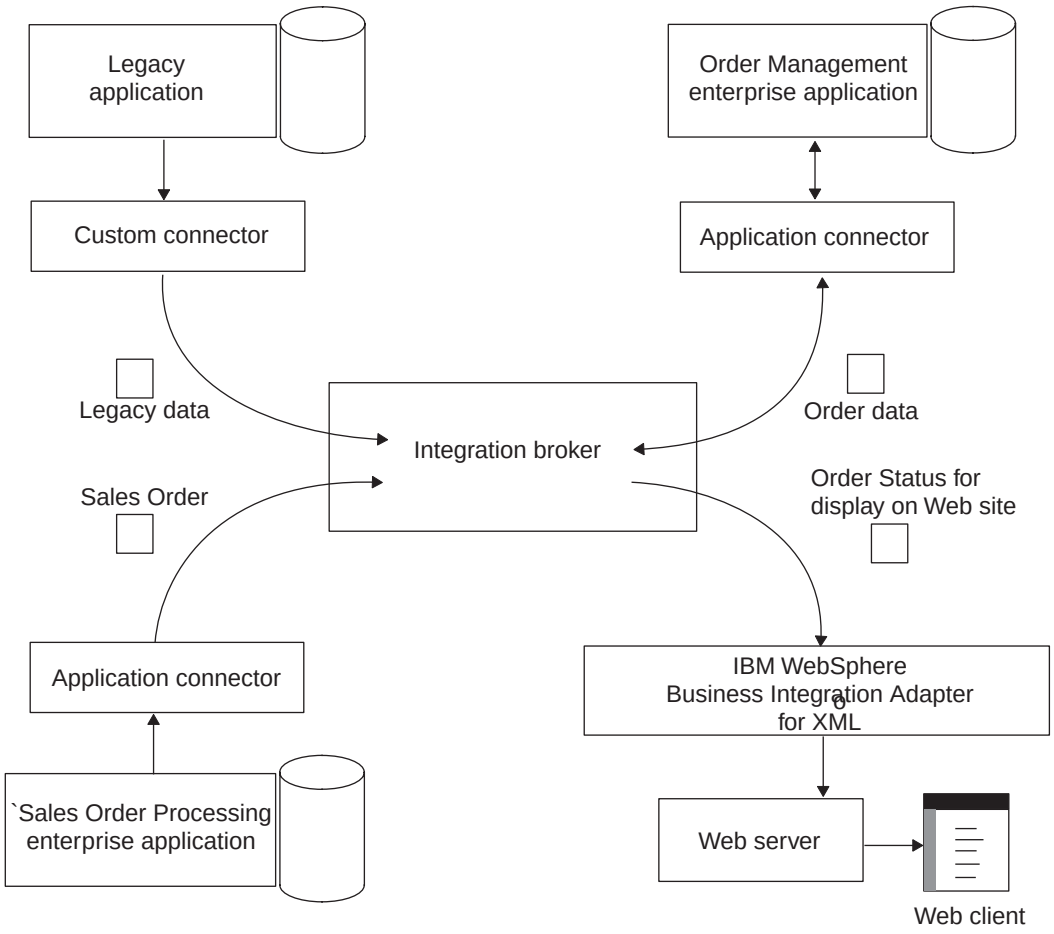

Figure 2. WebSphere business integration system

Business objects

As Table 2 shows, a business object is a two-part entity, consisting of a repository definition and a runtime object.

Table 2. Parts of a Business Object

Repository entity	Runtime object
Business object definition	Business object instance (often called a "business object")

Business object definition

A *business object definition* represents a group of attributes that can be treated as a collective unit. For example, a business object definition can represent an application entity and the operations that can be performed on the entity, such as create, retrieve, update, or delete. A business object definition can also represent other programmatic entities, such as the data contents of a business transaction form submitted from a Web browser. A business object definition contains *attributes* for each piece of data in the collective unit.

Note: For more information on the structure of a business object definition, see "Processing business objects" on page 106..

When you "develop a business object," you create a business object definition. You can create business objects definitions with the Business Object Designer tool, which provides an easy-to-use, graphical user interface (GUI) that allows you to define attributes of the business object. It supports saving the business object definition in the repository or in an external XML file.

Within Business Object Designer, you can create the business object definition in either of two ways:

- Manually, by using the dialogs of Business Object Designer to define attributes and other information for the business object definition.
- With an Object Discovery Agent (ODA), which automatically generates a business object definition by:
  - Examining specified entities within the application
  - "Discovering" the elements of these entities that correspond to business object attributes

Note: For information on how to use Business Object Designer to create business object definitions in either of these ways, see the *Business Object Development Guide*.

Business object instance

While the business object definition represents the collection of data, a business object instance (often just called a "*business object*") is the runtime entity that contains the actual data. For example, to represent a customer entity in your application, you can create a Customer business object definition that defines the information in the customer entity that needs to be sent to other applications. At runtime, the Customer business object, which is an instance of this business object definition, contains the information for a particular customer.

Connectors

The role of a *connector* is to send information about an application event to an integration broker or to receive information about a request from the integration broker.

WebSphere InterChange Server

When InterChange Server is the integration broker, a connector is a set of software modules and data maps that connect WebSphere Business Integration collaborations to an enterprise application or an external application. A *collaboration* represents a business process that can involve several applications. The connector acts as an intermediary for one or more collaborations, using an enterprise application's API, or some other program logic, to support a business process.

The information that the connector sends or receives is in the form of a business object. Therefore, each connector supports one or more business object definitions. These business object definitions have been designed to correspond to application data models or to other types of external entities. The business object closely reflects the data entity that it represents. Its structure and content match that of the entity.

WebSphere InterChange Server

When InterChange Server is the integration broker, the business integration system uses two kinds of business objects. The business object that a connector processes is called an *application-specific business object*. The business object that a collaboration processes is called a *generic business object*. For more information, see "Mapping services" on page 12..

Other integrator brokers

When a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Intregration Message Broker) or WebSphere Application Server is the integration broker, the business integration system uses a single kind of business object, the business object that a connector processes. Although this business object is an application-specific business object, the "application-specific" qualifier is often omitted because this is the only kind of business object used.

The connector uses information in its supported business object definitions to perform its major roles, as Table 3 shows.

Table 3. Operations on business objects for the different roles of a connector

Connector role	Operation on business object
"Event notification" on page 22	When an event that affects an application entity occurs (such as when a user of the application creates, updates, or deletes application data), a connector: • Creates a business object, based on the information in its business object definition • Fills this business object with data from an application entity • Sends this business object as an event to an integration broker
"Request processing" on page 24	When the integration broker requests a change to the connector's application or when the broker needs information from the connector's application, the connector: • Receives a business object from an integration broker • Uses information in the business object and its business object definition to create the appropriate application command that performs an operation • Sends any appropriate response information back to the integration broker

Note: Every connector *must* implement request processing. Implementation of event notification is optional (though does require some minor coding).

Connector components

The connector represents the application in the WebSphere business integration system, performing tasks in support of the application. For example, the connector polls the application for events and sends business objects that represent events to the integration broker. The connector also performs tasks in support of integration-broker requests, such as retrieving or modifying application data.

Figure 3 illustrates the components of a Java connector. The Java connector library is included in the generic services that the connector framework provides.

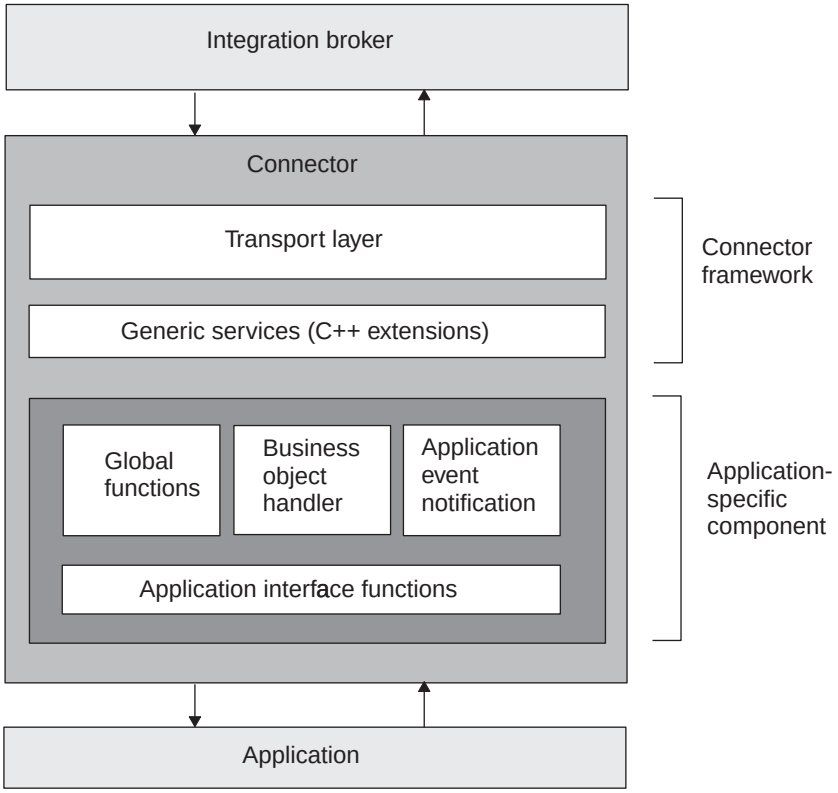

Figure 3. Components of a Java connector

As Figure 3 shows, a connector has the following components:

- "Connector framework"—Provided as part of the WebSphere Business Integration Adapters product to communicate with the integration broker.
- "Application-specific component" on page 19—Contains code you write to specify the actions of the application-specific tasks of the connector, such as basic initialization and setup methods, business object handling, and event notification.

Connector framework

The connector framework manages interactions between the connector and the integration broker. IBM provides this component to ease connector development. The connector framework is written in Java and includes a C++ extension to allow the development of the application-specific component in C++.

Other integration brokers

In an IBM WebSphere business integration system that uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker), or WebSphere Application Server as its integration broker, the connector framework is a nondistributed component; that is, it resides entirely on the adapter machine. Figure 4 shows the high-level connector architecture with the WebSphere message broker or WebSphere Application Server. For information on the connector architecture with InterChange Server as the integration broker, see "Connector controller" on page 10..

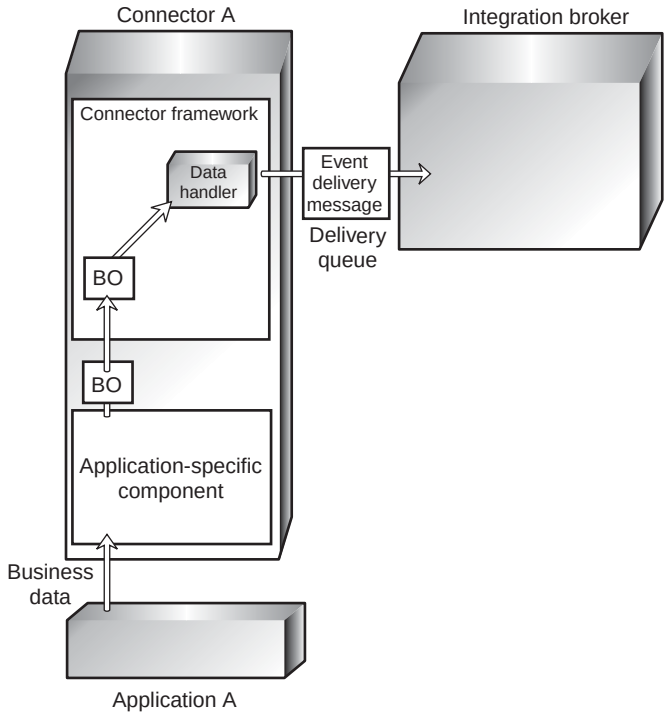

Figure 4. High-level connector architecture with a WebSphere message broker

The connector framework provides the services that Table 4 summarizes.

Table 4. Services of the connector framework

Component	Services
"Connector controller" on page 10 (InterChange Server only)	• Provides mapping between application-specific and generic business objects, and manages business object transfers between the connector and collaborations running in InterChange Server. • Provides other management services, such as monitoring the status of the connector

Table 4. Services of the connector framework (continued)

Component	Services
"Transport layer" on page 14	• Handles the exchange of business objects between the connector and the integration broker • Manages the exchange of startup and administrative messages between the connector controller and the client connector framework • Keeps a list of subscribed business objects
Java connector library on page "Java connector library" on page 19.	• Provides generic services to the application-specific component in the form of Java classes and methods

Connector controller

In an IBM WebSphere business integration system that uses InterChange Server as its integration broker, the connector framework is distributed to take advantage of services that InterChange Server provides. This distributed connector framework contains the following components:

- The *client connector framework* runs as part of the connector process on the client machine. It includes a transport layer, and the Java connector library. For more information on these components, see Table 4 on page 9..
- The *connector controller* runs within InterChange Server on the server machine.

Figure 5 illustrates the basic components of a connector within the InterChange Server system. InterChange Server, collaborations, and connector controllers run as a single process, and each connector runs as a separate process.

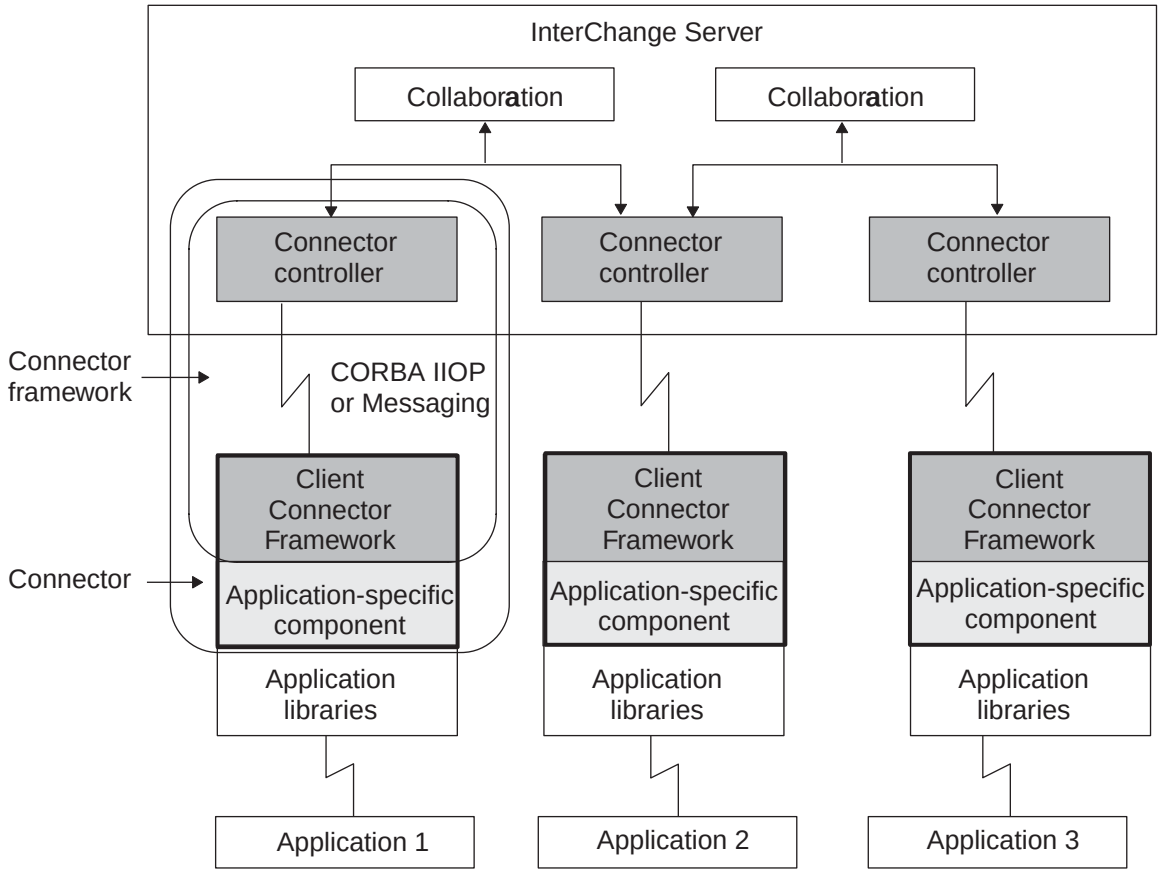

Figure 5. High-level connector architecture with WebSphere InterChange Server

The connector controller manages communication between the connector framework and collaborations. The primary type of information that connector components exchange is a business object. Other types of connector communication include startup information and administrative messages.

Note: A connector controller is instantiated by InterChange Server for each connector that has been defined in the InterChange Server repository. You do *not* need to provide code for the connector controller, as this component is internal to InterChange Server.

In addition to the features that the client connector framework provides, the connector controller provides the services that Table 5 summarizes.

Table 5. Services of the connector controller

Connector controller service	Description
"Mapping services" on page 12	The connector controller calls the map associated with each business object to transfer data between generic business objects and application-specific business objects.
"Business object subscription and publishing" on page 13	The connector controller manages collaboration subscriptions to business object definitions. It also manages connector queries about subscription status for a business object.

Table 5. Services of the connector controller (continued)

Connector controller service	Description
Service call requests (For more information, see "Initiating a request with InterChange Server" on page 25.)	The connector controller delivers collaboration service call requests to connectors. It also accepts return status messages and business objects from the connector and forwards them to InterChange Server.
Communication between components (For more information, see "Transport mechanism with InterChange Server" on page 15.)	The connector controller contains a transport driver to handle its side of the mechanism for exchanging business objects and administrative messages between the connector controller and client connector framework. It also performs remote-end synchronization to manages high-level synchronization between itself and the client connector framework. These services enable the connector controller to communicate with the connector, which might be installed remotely.

Note: The connector controller handles its own internal errors as well as errors from the client connector framework. In general, the connector controller logs exceptions and then evaluates whether further action is needed. When status messages are returned by the client connector framework, the connector controller forwards the messages to the collaboration.

Mapping services: The client connector framework sends and receives information in an application-specific business object. However, a collaboration generates information in a generic business object. Because application-specific business objects can differ from generic business objects, the InterChange Server system must convert business objects from one form to another so that data can be transmitted across the system. Data is transformed between generic and application-specific business objects by data *mapping*.

Data mapping converts business objects from generic to application-specific and from application-specific to generic forms. An application-specific business object closely reflects the data entity that it represents. Its structure and content match that of the entity. A generic business object, on the other hand, typically contains a superset of attributes that represents a typical, cross-application view of an entity's data. This type of business object is a composite of common information that many applications have about a particular entity. A generic business object serves as an intermediate point between data models.

Mapping is initiated by the connector and executed at runtime. For example, when a connector needs to map an application-specific business object to a generic business object, it runs an associated map to transfer data between the application-specific business object and the generic business object before sending the generic business object to a collaboration.

Mapping is handled by the connector controller. Figure 6 illustrates the connector in the InterChange Server system and shows the components of the connector.

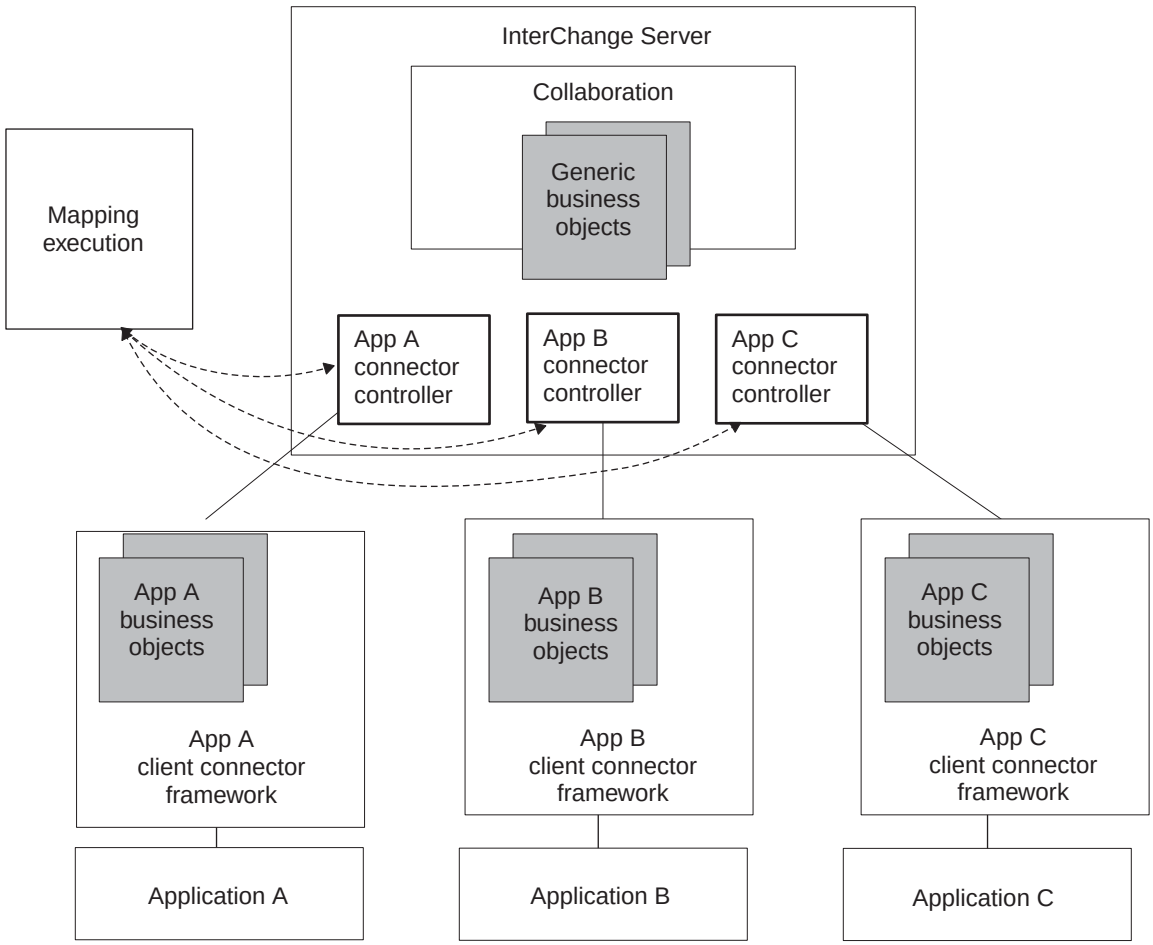

Figure 6. Mapping in the InterChange Server System

For more information on data mapping, refer to the *Map Development Guide* in the IBM WebSphere InterChange Server documentation set.

Business object subscription and publishing: Subscription handling is managed through a *subscription list*, which is a list of business objects to which collaborations have subscribed. Both the connector framework and the connector controller maintain a subscription list, as follows:

- The connector controller maintains a list of business objects to which collaborations have subscribed.

  When collaborations start, they subscribe to the business objects that they are interested in by informing the connector controller of their interest. The connector controller stores this information in a subscription list, which contains the name of the subscribing collaboration and the business object definition name and verb.

  When the connector controller receives a business object from the client connector framework, it checks its own subscription list to determine which collaborations have subscribed to this type of business object. It then forwards the business object to the subscribing collaboration.
- The connector framework also maintains a list of business objects to which collaborations have subscribed. However, this subscription list is a consolidated version of the connector controller's subscription list.

At initialization, the connector downloads its business object definitions and configuration properties from the InterChange Server repository. It also requests the subscription list from the connector controller. The subscription list that the connector controller sends to the client connector framework contains only the names of the business object definitions and verbs for these subscribed business objects. The connector framework stores this subscription list locally. Whenever a new collaboration starts up and subscribes to a business object, the connector controller notifies the connector framework so that the local subscription list is kept current.

As part of the initialization of the client connector framework, the connector framework instantiates a subscription manager. The *subscription manager* tracks all subscribe and unsubscribe messages that arrive from the connector controller and maintains a list of active business object subscriptions. Through the subscription manager, the application-specific connector component can query the connector framework to find out whether any collaborations are interested in a particular kind of business object.

Figure 7 illustrates the connector architecture for subscription handling.

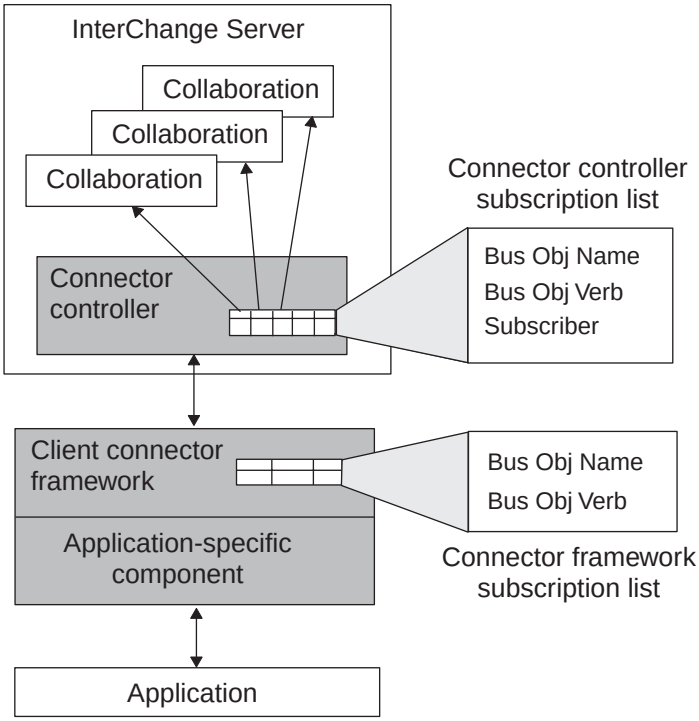

Figure 7. Subscription handling

For more information on subscriptions, see "Request processing" on page 24.

Transport layer

The transport layer of the connector framework handles the exchange of information between the connector and the integration broker. The transport layer of the connector framework provides the following services:

- Receives business objects from the integration broker and sends business objects to the integration broker:

Message service	Description
"Request processing" on page 24	Receives a business object from the integration broker and sends it to the application-specific component of the connector
"Event notification" on page 22	Receives a business object from the application-specific component of the connector and sends it to the integration broker

- Manages the exchange of startup and administrative messages between the connector and the integration broker.
- Keeps a list of business objects that are subscribed to

The transport mechanism of the transport layer depends on the integration broker in your business integration system:

- "Transport mechanism with InterChange Server"
- "Transport mechanism with other integration brokers" on page 18

Transport mechanism with InterChange Server: If the integration broker is InterChange Server (ICS), the transport layer handles the exchange of information between the connector controller, which resides within ICS, and the client connector framework.

Note: For more information, see "Connector controller" on page 10.

As Figure 8 shows, the transport layer for a connector that communicates with InterChange Server might include two transport drivers, one for the Common Object Request Broker (CORBA) and one for some message-oriented middleware (MOM).

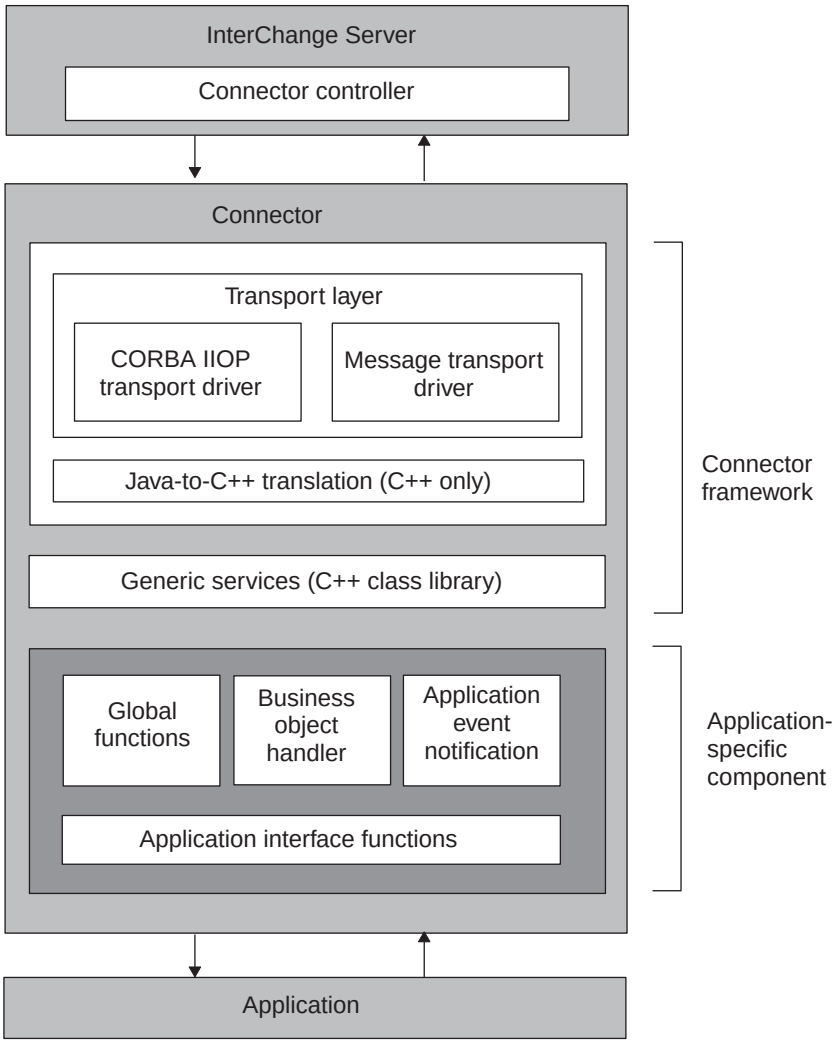

Figure 8. Connector architecture for communicating with InterChange Server

Table 6 summarizes the tasks that the transport layer performs and the transport mechanisms it can use.

Table 6. Tasks of the transport layer

Transport-layer task	Transport mechanism
Connector startup and exchange of startup messages between the connector controller and the client connector framework	CORBA
Administrative messages about the state of the client connector framework	CORBA
Sending business objects to the connector, initiated with a collaboration service call request	CORBA
Sending business objects from the connector, initiated with an event delivery	CORBA A message-oriented middleware system, including one of the following: • WebSphere MQ • Java Messaging Service (JMS)

This transport mechanism has the following tasks:

- At connector startup, the transport layer uses the Common Object Request Broker Architecture (CORBA) to transfer information from InterChange Server to the memory of the connector process.

  In the CORBA architecture, objects communicate through the Object Request Broker (ORB). The ORB is a set of libraries and services that connects an object, such as a connector controller, with another object, such as a client connector framework. The ORB enables objects to find each other at startup and to invoke methods on each other at runtime.

  With the ORB, the CORBA architecture provides a Naming Service that allows an object on the ORB to locate another object by name. At startup, the client connector framework uses the Naming Service to connect to the InterChange Server. The client connector framework then uses the ORB to request its application-specific connector configuration properties and its list of supported business object definitions from the repository. For more information, see "Starting up a connector" on page 63..

  Once the client connector framework and connector controller are active and connected, the client connector framework requests its list of business object subscriptions. At this point, connector initialization is complete, and the connector starts polling for events.
- For administrative messages about the state of the connector, the transport layer uses CORBA to send and receive state information for the connector controller.

  Changes in state of the client connector framework can be initiated from System Manager in the WebSphere Business Integration Toolset. Such changes include start, stop, pause, and resume operations, as well as retrieving the status. In addition, administrative messages can specify remote message logging.
- For sending business objects to the connector, initiated with a collaboration service call request, the transport layer also uses CORBA.

  CORBA technology includes the Internet Inter-ORB Protocol (IIOP) transport protocol. CORBA IIOP provides a lightweight, high-performance, synchronous communication mechanism that the connector controller and the client connector framework use to interact. Because the IIOP communication mechanism is synchronous, connector components can quickly determine whether a business object exchange was successful and can take appropriate action if necessary.
- For sending business objects from the connector, initiated with an event delivery, the connector can be configured to use either CORBA or a message-oriented middleware (MOM) system.

  When CORBA is used for business object subscription delivery, multiple business objects can be delivered concurrently, improving performance for subscription delivery. Using CORBA as a communication mechanism provides particularly good performance on a high-bandwidth LAN network.

  A messaging system provides asynchronous message delivery across a network, enabling connector components to send a message and continue processing without waiting for a response. The messaging system also provides persistent messaging, allowing the connector controller and client connector framework to operate independently.

  Note: In this case, connector components continue to use CORBA for startup and administrative messages.

  In the messaging communication mechanism, message transport is handled by transport drivers in the client connector framework and the connector controller. The message transport driver implements the low-level mechanism for exchanging data between InterChange Server and the underlying message

queuing software. Messages between the components of the connector are transported in a format defined by the messaging software.

This business integration system uses CORBA technology provided by the IBM Object Request Broker (ORB). Figure 9 illustrates the CORBA communication mechanism.

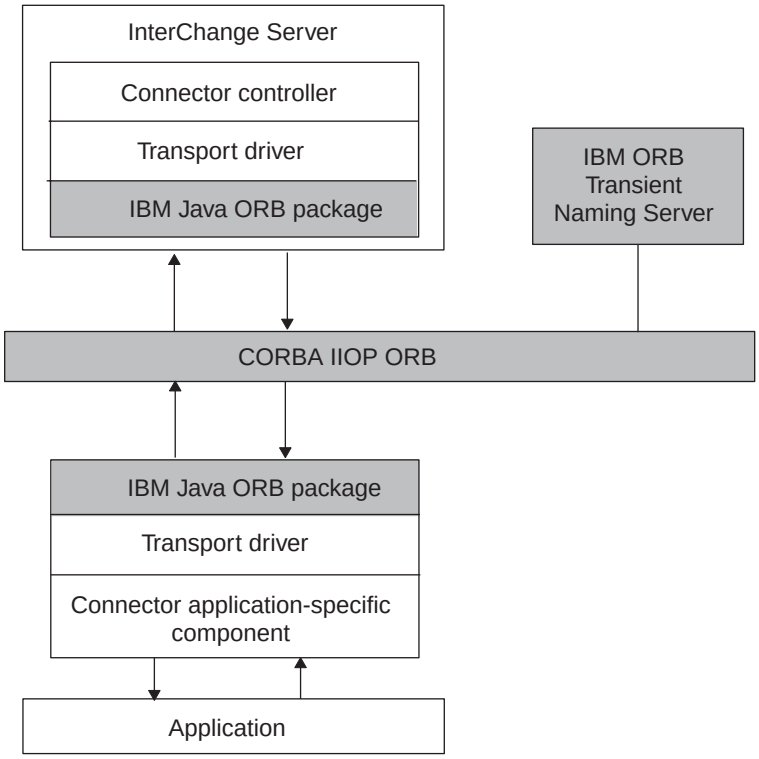

Figure 9. Communication within a connector using CORBA IIOP

Supported message-oriented middleware includes:

- IBM WebSphere MQ messaging suite. In this system, each active connector requires one unidirectional message queue. WebSphere MQ manages the queue using a queue manager. In this business integration system, each InterChange Server has one queue manager for all system components.
- Java Messaging Service (JMS)

Note: To configure a connector's transport mechanism for event delivery, set the `DeliveryTransport` standard property. For more information on this property, see Appendix A, "Standard configuration properties for connectors," on page 479.

Transport mechanism with other integration brokers: If the integration broker is a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the transport layer handles the exchange of information between the connector framework and the integration broker. The transport layer for a connector that communicates with the broker includes a single transport driver for the IBM WebSphere MQ messaging suite. Data is exchanged between applications by means of application-specific business objects, which are transported between the connector framework and the integration broker as

WebSphere MQ messages. The integration broker removes the message from the MQ queue, and passes it through the message flow for the queue.

This transport mechanism uses WebSphere MQ messages to perform the following tasks:

- For sending business objects *to* the connector, which initiates request processing, the transport layer converts the business object to an MQ message and puts this message onto the appropriate WebSphere MQ queue.
- For sending business objects *from* the connector, which initiates an event delivery, the transport layer takes the MQ message off the appropriate WebSphere MQ queue and converts it to an application-specific business object.

The connector framework uses a custom data handler to transform the application-specific business object to and from an MQ message of the appropriate wire format for the destination WebSphere MQ queue.

For more detailed information on the use of MQ messages and a connector, see the implementation guide for your integration broker.

Java connector library

The connector framework includes the Java connector library, which provides generic services and utilities for connector development. The primary services provided by the Java connector library are:

- Business object definition directory – Manages access to the business object definitions supported by a connector. Business object definitions are cached to improve connector performance in a distributed environment.
- Business object class – Provides methods for processing application information. This class allows the connector to handle application data in an object-oriented manner.
- Subscription manager – Enables the connector to check whether any collaborations are interested in a particular kind of business object.
- Logging utility – Enables the connector to post messages to the connector's standard output. Functionality includes configurable output destination and allows assigning error levels for all logged messages.
- Tracing utility – Enables the connector to generate trace messages for debugging purposes.

Note: For a summary of the C++ connector library and its classes, see 209Chapter 9, "Overview of the C++ connector library," on page 237.Java connector library and its classes, see Chapter 9, "Overview of the Java connector library," on page 233.

The Java connector library is a Java .jar file called `WBIA.jar`, which resides in the following directory:

`ProductDir``/lib`

Because Java is operating-system-independent, the Java connector library is available on all systems that the WebSphere Business Integration Adapters product supports

Application-specific component

The application-specific component of the connector contains code tailored to a particular application. This is the part of the connector that you design and code. The application-specific component includes:

- A connector base class to initialize and set up the connector
- A business object handler to respond to request business objects initialized by integration-broker requests
- If needed, an event notification mechanism to detect and respond to application events.

You develop your code for the application-specific component to use services provided by the connector framework. The connector class library provides access to these services. You can write your connector code in C++ or Java depending on the application programming interface (API) provided by the application.

If the application API is written in Java, you write the application-specific portion of the connector in Java, accessing services of the connector framework through the Java connector library.

Event-triggered flow

The Java connector library contain an API that allows a user-defined application-specific component to communicate with an integration broker through business objects. Applications can exchange information with other applications that the integration broker handles.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector can communicates with other applications through executing a collaboration. A *collaboration* represents a business process that can involve several applications. A connector transforms data and logic into a business object that carries information about an event in the connector's application. The business object triggers a collaboration business process and provides the collaboration with information that it needs for the business process.

Note: An external process can also initiate execution of collaborations through a call-triggered flow. For more information, see the *Access Development Guide* in the IBM WebSphere InterChange Server documentation set.

WebSphere Message Brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker), the connector might request information from or send information to other applications through WebSphere MQ workflows. The MQ workflow routes the information as appropriate.

When an event occurs in the application, the connector's application-specific component creates a business object to represent this event and sends the event to the integration broker. An *application event* is any event that affects an entity associated with a business object definition. To send an event to an integration broker, the connector initiates an *event delivery*. This event contains a business object. Therefore, the flow trigger that a connector initiates is called an *event-triggered flow* (see Figure 10).

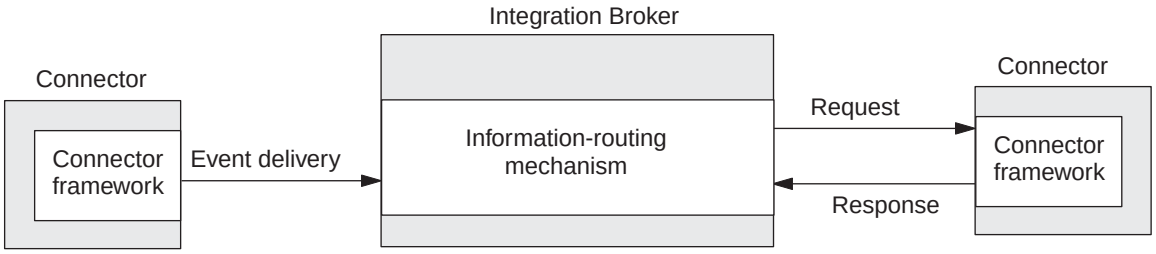

Figure 10. Event-triggered flow for WebSphere business integration system

Figure 10 shows event-triggering flow within the IBM WebSphere business integration system, which involves the following steps:

1. The connector creates the *triggering event*, which it sends to the integration broker during event delivery.

   When an event that affects an application entity occurs (such as when a user of the application creates, updates, or deletes application data), a connector creates a business object, which contains data from the application entity and a verb that indicates the operation performed on this data.
2. The application-specific component of the connector calls the `gotApplEvent()` method of the Java connector library to send the triggering event to the connector framework. Through this method call, the connector performs an *event delivery*, which initiates the event-triggered flow.
3. The connector framework performs any needed conversion of the triggering event to a business object, then sends this event to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector controller receives the triggering event, performing any needed mapping of the application-specific business object data to the appropriate generic business object. The connector controller then sends the triggering event to the specified collaboration to trigger its execution. This collaboration is one that has subscribed to the business object that the event represents. The collaboration receives this business object in its incoming port.

4. The integration broker uses whatever logic it provides to route the event to the appropriate application. If it is so programmed, it might perform a *request*, routing the event information to the connector of some destination application, which would receive the event containing its request business object. In addition, this destination connector might send a request *response* back to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the collaboration might perform a *service call request* to send a business object to the connector controller of the destination connector, which is bound to its outgoing port. This connector controller performs any needed conversion from the resulting generic business object to the appropriate application-specific business object. It then performs a *service call response* to send the event response to the connector controller, which routes it back to the collaboration.

As Figure 10 shows, a connector can participate in one of two roles:

- "Event notification"—the connector sends an event (in the form of a business object) to the integration broker to notify it of some operation that has occurred in the application.
- "Request processing" on page 24—the connector receives a request business object from an integration broker.

Each of these connector roles is described in more detail in the following sections.

Event notification

One role of a connector is to detect changes to application business entities. When an event that affects an application entity occurs, such as when a user of the application creates, updates, or deletes application data, a connector sends an event to the integration broker. This *event* contains a business object and a verb. This role is called *event notification*.

This section provides the following information about event notification:

- "Publish-and-subscribe model"
- "Event-notification mechanism"

Publish-and-subscribe model

A connector assumes that the business integration system uses a *publish-and-subscribe model* to move information from an application to an integration broker for processing:

- An integration broker *subscribes* to a business object that represents an event in an application.

> **WebSphere InterChange Server**
> If your business integration system uses InterChange Server, a collaboration *subscribes* to a business object that represents an event in an application, and then the collaboration waits.

- A connector uses an event-notification mechanism to monitor when application events occur. When an application event does occur, the connector *publishes* a notification of the event in the form of a business object and a verb. When the integration broker receives an event in the form of the business object that it has subscribed to, it can begin the associated business logic on this data.

> **WebSphere InterChange Server**
> If your business integration system uses InterChange Server, the connector controller checks its own subscription list when it receives a business object from the connector framework to determine which any collaborations have subscribed to this type of business object. If so, it then forwards the business object to the subscribing collaboration. When a collaboration receives the subscribed event, it begins executing.

Event-notification mechanism

An *event-notification mechanism* enables a connector to determine when an entity within an application changes. When an event occurs in an application, the

connector application-specific component processes the event, retrieves related application data, and returns the data to the integration broker in an business object.

Note: This section provides an introduction to event notification. For more information on how to implement an event-notification mechanism, see Chapter 5, "Event notification," on page 115.

The following steps outline the tasks of an event-notification mechanism:

1. An application performs an event and puts an event record into the event store.

   The *event store* is a persistent cache in the application where event records are saved until the connector can process them. The *event record* contains information about the change to an event store in the application. This information includes the data that has been created or changed, as well as the operation (such as create, delete, or update) that has been performed on the data.
2. The connector's application-specific component monitors the event store, usually through a polling mechanism, to check for incoming events. When it finds an event, it retrieves its event record from the event store and converts it into an application-specific business object with a verb.
3. Before sending the business object to the integration broker, the application-specific component can verify that the integration broker is interested in receiving the business object.

   WebSphere InterChange Server

   If your business integration system uses InterChange Server, the connector framework does *not* assume that the integration broker is always interested in every supported business objects. At initialization, the connector framework requests its subscription list from the connector controller. At runtime, the application-specific component can query the connector framework to verify that some collaboration subscribes to a particular business object. The application-specific connector component can send the event *only* if some collaboration is currently subscribed. The application-specific component sends the event, in the form of a business object and a verb, to the connector framework, which in turn sends it to the connector controller within ICS. For more information, see "Mapping services" on page 12.

   Other integration brokers

   If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework assumes that the integration broker is interested in *all* the connector's supported business objects. If the application-specific connector component queries the connector framework to verify whether to send the business object, it will receive a confirmation for *every* business object that the connector supports.
4. If the integration broker is interested in the business object, the connector application-specific component sends the event, in the form of a business object and a verb, to the connector framework, which in turn sends it to the integration broker.

Figure 11 illustrates the components of the event-notification mechanism. In event notification, the flow of information is from the application to the connector and then to the integration broker.

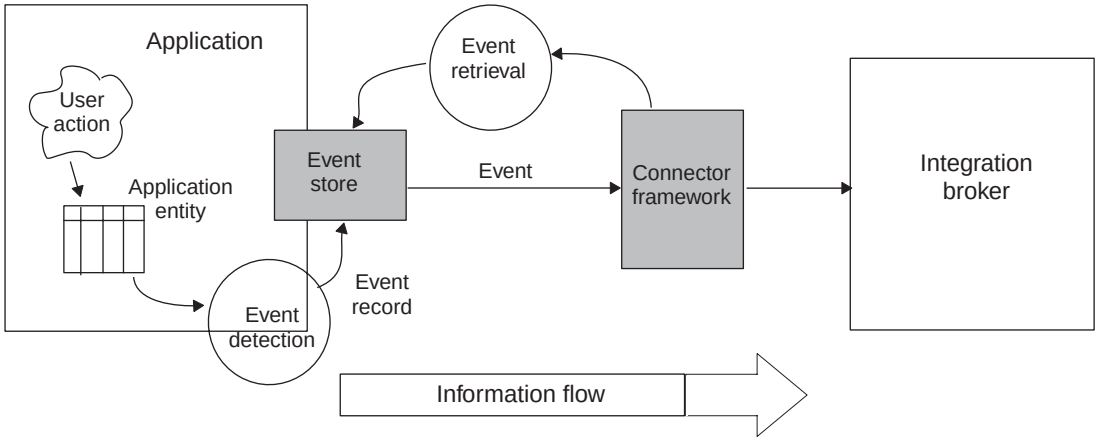

Figure 11. Event detection and retrieval

Request processing

In addition to detecting application events, another role of a connector is to respond to requests from the integration broker. A connector receives a *request business object* from a integration broker when the broker requests a change to the connector's application or needs information from the connector's application. In general, connectors perform create, retrieve, and update operations on application data in response to requests from a collaboration. Depending on the application's policies, the connector might also support delete operations. This role is called *request processing*.

WebSphere InterChange Server

If your business integration system uses InterChange Server, request processing can sometimes be called "service call request processing". The connector receives a business object from its connector controller, which receives it from a service call of a collaboration.

Note: This section provides an introduction to request processing. For more information on how to implement request processing in your connector, see Chapter 4, "Request processing," on page 79.

Request processing involves the following steps:

1. As Figure 10 on page 21 shows, an integration broker initiates request processing by sending a request to the connector framework. This request is in the form of a business object, called the *request business object*, and a verb. For more information, see "Initiating a request" on page 25.
2. The connector framework has the task of determining which *business object handler* in the application-specific component should process the request business object. For more information, see "Choosing a business object handler" on page 25.
3. The connector framework passes the request business object to the business object handler defined for it in its business object definition.

The connector framework does this by calling the `doVerbFor()` method defined in the business object class and passing in the request business object. The business object handler then processes the business object, converting it to one or more application requests.

4. When the business object handler completes the interaction with the application, it returns a return-status descriptor and possibly a response business object to the connector framework. For more information, see "Handling a request response" on page 26.

Initiating a request

The way a request is initiated depends on the integration broker in your IBM WebSphere business integration system:

- "Initiating a request with InterChange Server"
- "Initiating a request with other integration brokers"

Initiating a request with InterChange Server: If your business integration system uses InterChange Server, the collaboration initiates a *service call request*, sending the request over one of its collaboration ports. When you bind a port of a collaboration object, you associate the port with a connector (or another collaboration object). Collaboration ports enable communication between bound entities, so that the collaboration object can accept the business object that triggers its business processes, and then send and receive business objects as service call requests and responses.

Note: For more information on how to define collaboration ports, see the *Collaboration Development Guide*. For information on how to bind ports of a collaboration object, see the *Implementation Guide for WebSphere InterChange Server*. Both these documents are in the IBM WebSphere InterChange Server documentation set.

One the service call request is initiated, the InterChange Server system takes the following steps:

1. The connector controller for the connector bound to the collaboration port receives the service call request. If necessary, the connector controller maps the generic business object to an application-specific business object before sending the request to the connector framework.
2. The connector controller forwards the service call request to the connector framework. The connector controller sends the request business object as a Java object.

Initiating a request with other integration brokers: If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the integration broker initiates a request by sending a message to the WebSphere MQ queue associated with the connector. One the request is initiated, the connector framework gets the WebSphere MQ message off using its transport layer and converts the message to the appropriate business object using a custom data handler.

For more information on the IBM WebSphere business integration system and request processing, see the implementation guide for your integration broker.

Choosing a business object handler

A *business object handler* is the Java class that is responsible for transforming the request business object into a request for the appropriate application operation. An

application-specific component includes one or more business object handlers to perform tasks for the verbs in the connector's supported business objects. Depending on the active verb, a business object handler can insert the data associated with a business object into an application, update an object, retrieve the object, delete it, or perform another task.

Based on this response business object's business object definition, the connector framework obtains the correct business object handler for the associated business object:

- When the connector starts up, the connector framework receives from the connector controller the list of business objects that the connector supports.
- The connector framework calls the `getConnectorBOHandlerForBO()` method (defined in the connector base class) to instantiate one or more business object handlers.
- For each supported business object, the `getConnectorBOHandlerForBO()` method returns a reference to a business object handler, and this reference is stored in the business object definition in the memory of the connector process.

All conversions between business objects and application operations take place within the business object handler (or handlers).

For more information about how to implement the `getConnectorBOHandlerForBO()` method, see "Obtaining the business object handler" on page 66..

Handling a request response

Once a connector has processed this request and completed the interaction with the application, it can return a response to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework returns a *service call response* to the collaboration. Using information in the return-status descriptor, the collaboration can determine the state of its service call request and take appropriate actions.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework's response includes:

- A status indicator, which contains the information return-status descriptor
- Any business object messages, which contain the optional response business objects

The connector framework puts this response information onto the connector's queue. However, for the message transport to be synchronous (that is, for some program to wait for a response), a program must post its request message to the integration broker on a synchronous request queue and expect its response from the broker on a synchronous response queue. A correlation ID on the response message identifies the message request to which it is responding.

Tools for adapter development

In the IBM WebSphere business integration system, the *connector* is a component of a WebSphere Business Integration adapter. As discussed in "Adapters in the WebSphere business integration system" on page 3, an *adapter* includes runtime components to support communication between an integration broker and applications or technologies. The adapter also includes an *adapter framework*, which includes components for the configuration, runtime, and development of custom adapters in cases where a prebuilt adapter for a particular legacy or specialized application is not currently available as part of the WebSphere Business Integration Adapters product.

The adapter framework includes configuration tools that assist in the development of the adapter components listed in Table 7..

Table 7. Adapter framework support for the development of a connector

Adapter component	Configuration tool	API
Business object	Business Object Designer	Not applicable
Object Discovery Agent (ODA)	Business Object Designer	Object Discovery Agent Development Kit (ODK)
Connector	Connector Configurator	Java Connector Library

In addition to the adapter framework, the WebSphere Business Integration Adapters product also provides the *Adapter Development Kit* (ADK). The ADK is a toolkit that provides code samples of connectors, ODAs, and data handlers. For more information, see "Adapter Development Kit" on page 28.

Development support for business objects

Table 8 shows the tools that the WebSphere Business Integration Adapters product provides to assist in the development of business objects.

Table 8. Development tools for business object development

Development tool	Description
Business Object Designer	Graphical tool that assists in the creation of business object definitions, either manually or through an ODA.

For a brief introduction to business objects, see "Business objects" on page 5. For more information on the use of the Business Object Designer, see the *Business Object Development Guide*.

Development support for ODAs

Table 8 shows the tools that the WebSphere Business Integration Adapters product provides to assist in the development of an ODA.

Table 9. Development tools for ODA development

Development tool	Description
Business Object Designer	Graphical tool that assists in the creation of business object definitions, either manually or through an ODA.
Object Discovery Agent Development Kit (ODK)	Set of Java classes with which you can create a custom ODA.

In addition, the ADK provides sample ODAs in the following product subdirectory:

`DevelopmentKits\Odk`

For a brief introduction to ODAs, see "Business objects" on page 5. For more information on the use of the Business Object Designer and the development of ODAs, see the *Business Object Development Guide*.

Development support for connectors

Table 10 shows the tools that the WebSphere Business Integration Adapters product provides to assist in the development of connectors.

Table 10. Development tools for connector development

Development tool	Description
Connector Configurator	Graphical tool that assists in the configuration of the connector
Adapter Development Kit	Includes sample code for Java connectors and ODAs

The supported operating-system environment for connector development is Windows 2000. Connectors can be written in either C++ or Java, depending on the language of your application API.

Connector Configurator

Connector Configurator is a graphical tool that allows you to configure a connector. It provides the ability to set the following information:

- Connector configuration properties
- Supported business objects
- Associated maps (with InterChange Server only)
- Log and message files
- Data-handler configuration (for guaranteed event delivery)

This graphical tool runs on Windows 2000 and Windows XP. Therefore, these platforms are for connector configuration.

Note: For more information on the use of Connector Configurator, see Appendix B, "Connector Configurator," on page 495.

Adapter Development Kit

The Adapter Development Kit (ADK) provides files and samples to assist in the development of an adapter. It provides samples for many of the adapter components, including an Object Discovery Agent (ODA), a connector, and a data handler. The ADK provides these samples in the `DevelopmentKits` subdirectory of your product directory.

Note: The ADK is part of the WebSphere Business Integration Adapters product and it requires its own separate Installer. Therefore, to have access to the development samples in the ADK, you must have access to the WebSphere Business Integration Adapters product and install the ADK. Please note that the ADK is available *only* on Windows systems.

Table 11 lists the samples that the ADK provides for the development of a connector, as well as the subdirectory of the `DevelopmentKits` directory in which they reside.

Table 11. ADK samples for connector development

Adapter Development Kit component	Description	DevelopmentKits subdirectory
Java Connector Development Kit (JCDK)	Provides sample code for a Java connector.	`jcdk` `edk\ConnectorAgent`
Twineball adapter sample	Provides a sample adapter, which includes a connector.	`Twineball_sample`

The ADK provides an adapter sample in the `Twineball_sample` subdirectory of `DevelopmentKits`. This sample contains several components of an adapter, including a connector, a data handler, and an Object Discovery Agent (ODA). For more information, see the *Adapter Development Kit Samples Guide*.

Connector Development Kit: The ADK includes the Java Connector Development Kit (JCDK), which provides components for use in the development of a connector. The components of the JCDK reside in the following *ProductDir*`\DevelopmentKits` subdirectory:

```
DevelopmentKits\jcdk
```

Table 12 describes the contents of the subdirectories in the jcdk directory.

Table 12. Components of the Connector Development Kit

Connector Development Kit component	Description	Subdirectory
Code samples	Sample code for a simple low-level Java connector	`samples`

The JCDK includes the following code samples to help in the development of your Java connector written with the low-level Java connector library:

```
DevelopmentKits\jcdk\samples
```

In addition, the JCDK includes code samples for a Java connector written with the Java connector library in the following directory:

```
DevelopmentKits\edk\ConnectorAgent
```

To compile a Java connector, use the Java compiler provided with the IBM Java Developers Kit (JDK). For more information, see "Compiling the connector" on page 210.

Note: The WebSphere Business Integration Adapters product also provides a C++ version of the Connector Development Kit for use in development connectors in the C++ programming language. For more information, see the *Connector Development Guide for C++*.

ODA samples: The Adapter Development Kit includes samples for an Object Discovery Agent (ODA). These samples reside in the following directory:

```
DevelopmentKits\Odk
```

For more information, see "Development support for ODAs" on page 27.

Overview of the connector development process

This section provides an overview of the connector development process, which includes the following high-level steps:

1. Install and set up the IBM WebSphere business integration system software and install the Java Development Kit (JDK).
2. Design and implement the connector.

Setting up the development environment

Before you start the development process, the following must be true:

- The IBM WebSphere business integration system software is installed on a machine that you can access.

  WebSphere InterChange Server

  If your business integration system uses InterChange Server, refer to the *System Installation Guide for UNIX* or *for Windows* (in the WebSphere InterChange Server documentation set) for information on how to install and start up the InterChange Server system.

  Other integration brokers

  If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere Integrator Broker, WebSphere Business Integration Message Broker), refer to the installation chapter of the *Implementing Adapters for WebSphere Message Brokers* for information on how to install and start up the IBM WebSphere business integration system. If your business integration system uses WebSphere Application Server, refer to the installation chapter of the *Implementing Adapters for WebSphere Application Server* for information on how to install and start up the IBM WebSphere business integration system.

- The Java Development Kit (JDK) 1.3.1 or a JDK-compliant development product is installed on the development machine.

  The Java compiler is part of the JDK. Therefore, the JDK must be installed for you to be able to create a new connector:

  – For Windows platforms, the IBM JDK is provided on the product CD. However, the product Installer does *not* automatically install it on your system. For more information about how to install the JDK as part of the InterChange Server product, see the *System Installation Guide for Windows*. For information on how to install it as part of the WebSphere Business Integration Adapters product, see the *WebSphere Business Integration Adapters Installation Guide*.

  – For UNIX platforms, you must download the JDK from a website and install it on your system. For more information about how to install the JDK as part of the InterChange Server product, see the *System Installation Guide for UNIX*. For information on how to install it as part of the WebSphere Business Integration Adapters product, see the *WebSphere Business Integration Adapters Installation Guide*

- Ensure that the development environment can access the directories that contain the connector library files. To compile the connector, the compiler *must* be able to access the connector library.

For information on compiling a connector, see "Compiling the connector" on page 210.

InterChange Server

- If your business integration system uses InterChange Server, the InterChange Server repository's database server and ICS are running.

  Note: This step is required only when you are ready to configure the connector with Connector Configurator. For development only, you can create the connector class, without connecting to ICS.

  For an overview of how to configure a connector, see Chapter 8, "Adding a connector to the business integration system," on page 209. For information on starting up the IBM WebSphere business integration system, see your system installation guide.

End of InterChange Server

Note: To create a connector, you do not need to run the messaging software. However, the messaging software must be running before you can execute and test the connector.

Stages of connector development

As part of the connector development process, you code the application-specific component of the connector and then compile and link the connector source files. In addition, the overall process of developing a connector includes other tasks, such as developing application-specific business objects. Here is an overview of the tasks in the connector development process:

1. Identify the application entities that the connector will make available to other applications, and investigate the integration features provided by the application.

InterChange Server

2. If your business integration system uses InterChange Server, identify generic business objects that the connector will support, and define application-specific business objects that correspond to the generic objects.
3. If your business integration system uses InterChange Server, analyze the relationship between the generic business objects and the application-specific business objects, and implement the mapping between them.

End of InterChange Server

4. Define a connector base class for the application-specific component, and implement functions to initialize and terminate the connector.
5. Define a business object handler class and code one or more business object handlers to handle requests.
6. Define a mechanism to detect events in the application, and implement the mechanism to support event subscriptions.
7. Implement error and message handling for all connector methods.
8. Build the connector.
9. Configure the connector.

WebSphere InterChange Server

If your business integration system uses InterChange Server, use Connector Configurator to create the connector definition and save it in the InterChange Server repository. You can call Connector Configurator from System Manager.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, use Connector Configurator to define and create the connector configuration file.

10. If WebSphere MQ will be used for messaging between connector components, add message queues for the connector.
11. Create a startup script for the new connector.
12. Test and debug the connector, recoding as necessary.

Figure 12 provides a visual overview of the connector development process and provides a quick reference to chapters where you can find information on specific topics. Note that if a team of people is available for connector development, the major tasks of developing a connector can be done in parallel by different members of the connector development team.

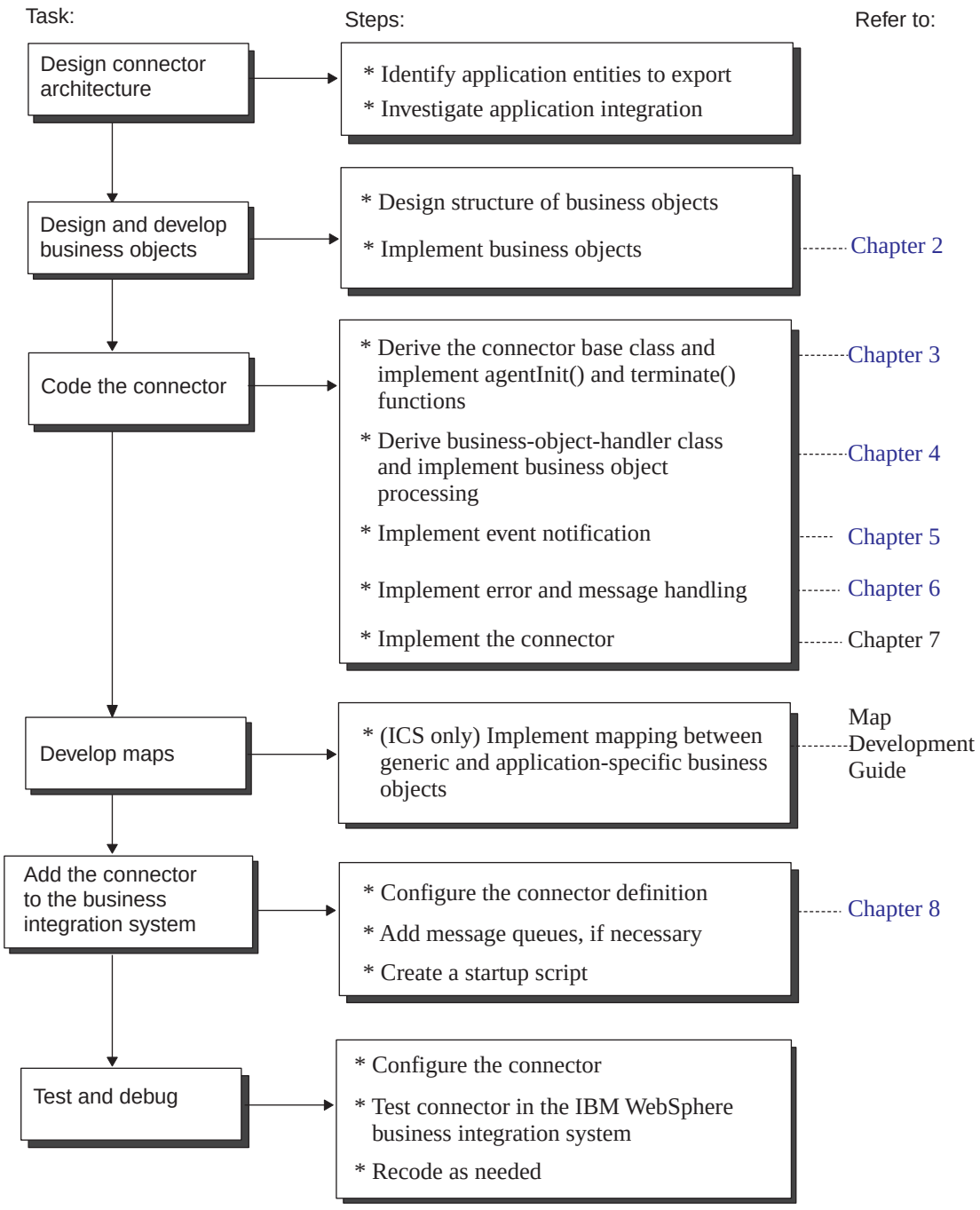

Figure 12. Overview of the Java Connector development process

Part 2. Building a connector

Chapter 2. Designing a connector

This chapter provides an overview of analysis and design issues to consider when planning a connector development project. The chapter presents topics that can help you judge the complexity of building a connector for your application or technology.

As with most software development projects, careful planning early in the connector development cycle helps prevent problems during later implementation phases. This chapter contains the following sections:

Scope of a connector development project

IBM provides a *connector framework* as part of the Java Connector Development Kit. The connector framework contains all the code necessary for the connector to interact with an integration broker and provides a basic infrastructure for interaction with the application.

Your task as a connector developer is to code the application-specific component of a connector, and if necessary, develop the event notification mechanism. The complexity of the design for your connector and the time required for the connector's implementation will vary based on the application.

To understand the scope and complexity of a connector development project, you may want to develop a project plan before beginning a new connector. As you develop the project plan, you need to identify the business requirements for the connector, define the application data that the connector will handle, and determine what application business processes the connector and business objects will work with. Developing a project plan can help you understand application functionality in the areas of business objects, business object processing, and event management.

Working through the topics in this chapter can help you estimate the time and effort needed to complete the connector development task. Each topic provides a set of questions that are intended to develop understanding of specific aspects of an application that might increase or decrease the complexity of the connector development task. A complete set of answers to the questions for each topic provides a high-level architecture for your connector.

Step in connector design	For more information
Obtain information about the application that is relevant to the design of the connector architecture.	"Designing the connector architecture" on page 38

Step in connector design	For more information
Ensure that application-specific business objects adequately represent the application entities that the connector needs to export.	"Designing application-specific business objects" on page 43
Design the event notification mechanism so that the application can notify the connector of relevant events.	"Event notification" on page 51

Designing the connector architecture

To design the connector architecture, consider evaluating the following areas of the application that the connector is to support:

- "Understanding the application environment" on page 39
- "Determining connector directionality" on page 40
- "Getting data in and out of the application" on page 41

The specific areas within an application that affect connector design are illustrated in Figure 13.. In this figure, the clouds show the high-level tasks required for connector development.

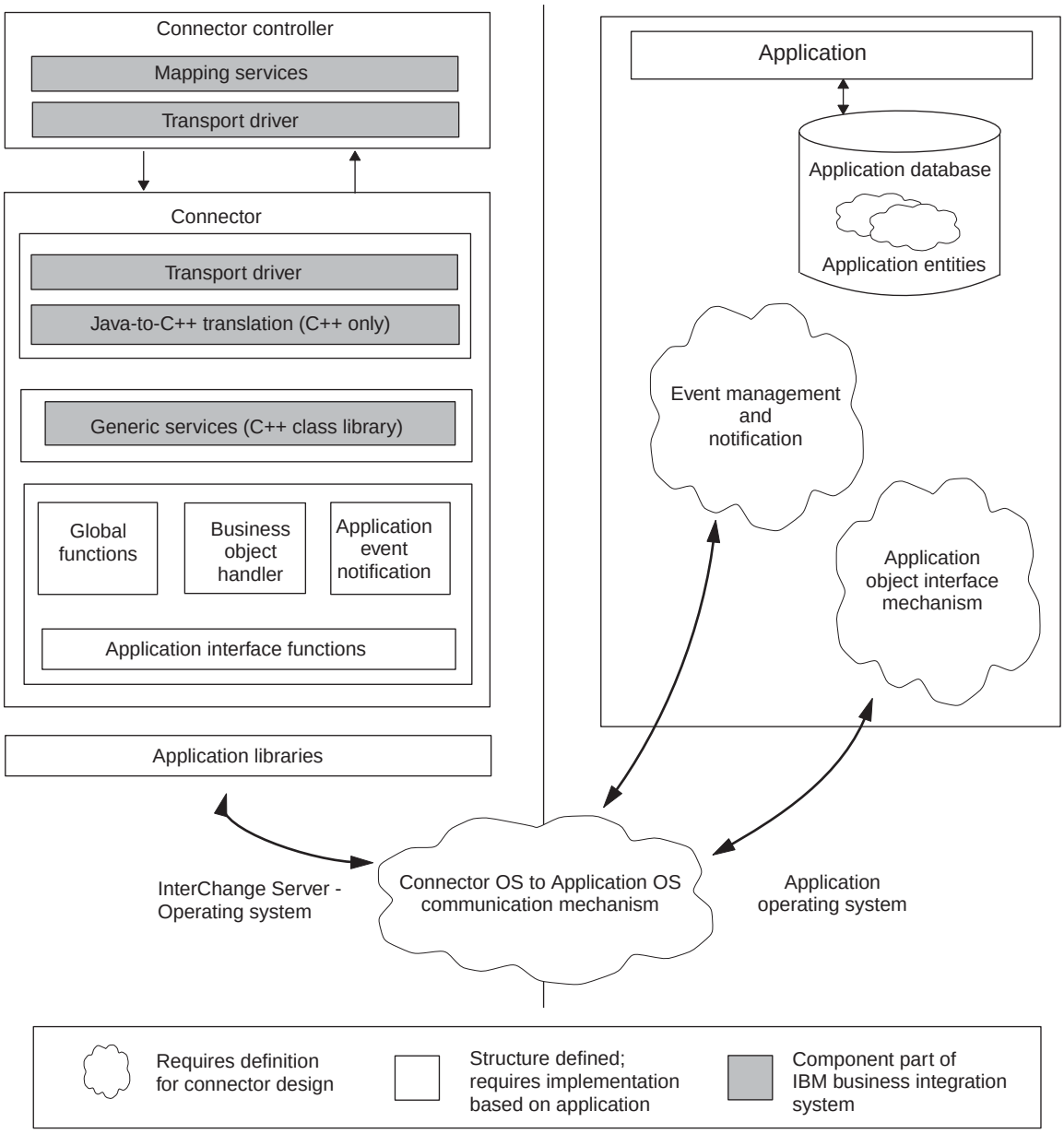

Figure 13. Areas of an application that affect connector design

Understanding the application environment

Understanding the application environment is the first step in assessing the feasibility of a connector development project. To obtain an understanding of the aspects of an application that affect connector development, consider these topics and questions:

Operating system

- What operating system does the application run on?

Programming languages

- What programming languages were used to create the application?

Application execution architecture

- What is the execution architecture of the application? For example, in a centralized architecture, the application and its database might both reside on a mainframe system. In this case, both application processing and database processing occur on this central system.

  Alternatively, in a client-server architecture, the database might reside on a server, and the application front-end program might be a client running on another machine, such as a personal computer. Other types of application execution architecture are online transaction processing and file server architecture.

Database type

- Is there a central database for application data? If application data is stored in a central database, what type of database is it? Example database types are RDMS and flat file.

Distributed application

- Is the application distributed across multiple servers?
- Is the application database distributed across multiple servers?

During project assessment, you may want to identify and work with an application expert. This person can also provide assistance during business object development and connector development.

Determining connector directionality

Early on in the project planning phase, you need to determine what roles the connector will perform for the application:

- Request processing—Update application data at the request of an integration broker. For more information, see "Request processing" on page 24..
- Event notification—Detect application events and send notification of events to the integration broker. For more information, see "Event notification" on page 22..

These roles determine the *directionality* that the connector supports:

- Unidirectional— some connectors might need to operate in only one direction, passing data from the application to the integration broker, or from the integration broker to the application.
  - To inform an integration broker that changes have occurred in the application, a connector must support event notification.
  - To receive data from an integration broker, a connector must support request processing, in which it interacts with the application to support Create, Retrieve, Update, or Delete operations as requested by the integration broker.

  For example, a connector might simply need to receive request business objects from an integration broker and pass them to an application. The connector for an application that serves only as the destination is a unidirectional connector – it implements request handling to pass data to the application, but it does not implement event notification. Knowing early in the development cycle that your connector will operate unidirectionally can save a significant amount of development time.
- Bidirectional—most connectors need to operate in *both* directions, passing data from the application to an integration broker *and* receiving data back from the integration broker.

To be bidirectional, your connector needs to support *both* event notification and request processing.

For information on how to provide event notification support in your connector, see Chapter 5, "Event notification," on page 115.

Getting data in and out of the application

An important aspect of the connector development project plan is to determine how the connector will get data into and out of the application. Ideally, an application provides an application programming interface (API) that includes all of the following features:

- Support for Create, Retrieve, Update, and Delete (CRUD) operations at the object level
- Encapsulation of all of the application business logic
- Support for delta and after-image operations
- An event-management strategy that allows external notification at the subobject level.

Typically, however, an application interface falls short of this ideal.

In your project plan, you need to establish whether a formal application API exists and evaluate its robustness, or, if an API does *not* exist, determine whether there is a suitable workaround. Keep in mind that an application CRUD interface can be anything from batch file imports and extracts to a COM/DCOM server, so be sure to explore all possible avenues. Refer to the application business object scope specified in Table 13 when exploring the application object CRUD interface.

Consider the following tasks:

- "Examining previous integration efforts"—Have there been any other efforts to integrate with this application?
- "Determining whether application data is shared with other applications" on page 42—Is the application data shared by other applications?
- "Examining an application API" on page 42—Is there an existing mechanism that the connector can use to communicate with the application?
- "Application use of batch clean-up or merge programs" on page 43—Does the application use batch clean-up or merge programs?

These questions are discussed in more detail in the following sections.

Examining previous integration efforts

If you have access to previous efforts to integrate other applications with your application, you might be able to find ways of getting data into and out of the application. Even if you decide to implement another approach to application integration, the previous integration effort may provide useful design information.

When examining previous integration efforts, consider these questions:

- What was the purpose of the integration?
- Does the integration use interfaces that modify or retrieve information from the application? If so, describe the mechanism used to modify or retrieve information.
- If the integration can process an event generated in the application, what is the mechanism used to trigger event processing?
- What is the mode of the existing integration? (batch, asynchronous, and so on)

- Will your connector replace the pre-existing integration? If not, will previous integrations work with the data entities that your connector will be working with?

In your answers, include information on all previous integration efforts that interact with the application in different ways.

Determining whether application data is shared with other applications

Your application might be one of several applications creating or updating data in a single database. In this case, your connector might have to consider an application data entity based on work that other applications are also doing. If you determine that your connector will be sharing application data with other applications, consider these questions:

- What is the mechanism used by the other applications to gain access to the application data?
- Do other applications create, retrieve, update, or delete application data? If so, what mechanism do other applications use for each verb?
- Is there object-specific business logic used by other applications? Is the logic consistent throughout all of the applications?

Provide answers to these questions for all applications that share the application data.

Examining an application API

If the application provides an API or other mechanism that the connector can use to communicate with the application, examine the API and review any available documentation. Keep in mind the following questions about the API:

- Does the API allow access for Create, Retrieve, Update, and Delete operations?
- Does the API provide access to all attributes of a data entity?
- Are there inconsistencies in the API implementation? Is the navigation to Create/Retrieve/Update/Delete the same regardless of the entity?
- Describe the transaction behavior of the API. For example, an API might simply enable the connector to run a report, which the connector can then read and use for processing. Or the API might be more robust, providing ways of performing asynchronous or synchronous Create and Update operations.
- Does the API allow access to the application for event detection? For example, if an application event-notification mechanism uses a database table as an event store, does the API allow access to this table?
- Is the API suited for metadata design? APIs that are forms-based, table-based, or object-based are good candidates. For information on metadata design, see "Assessing support for metadata-driven design" on page 47.
- Does the API enforce application business rules? In other words, is it an API that interacts at the table level, form level, or object level?

The recommended approach to connector development is to use whatever API the application provides. The use of an API helps ensure that connector interactions with the application abide by application business logic. In particular, a high-level API is usually designed to include support for the business logic in the application, whereas a low-level API might bypass application business logic.

As an example, a high-level API call to create a new record in a database table might evaluate the input data against a range of values, or it might update several

associated tables as well as the specified table. Using SQL statements to write directly to the database may bypass the data evaluation and related table updates performed by an API.

If no API is provided, the application might allow its clients to access its database directly using SQL statements. If you use SQL statements to update application data, work closely with someone who knows the application well so that you can be sure that your connector will not bypass application business logic.

This aspect of the application has a major impact on connector design because it affects the amount of coding that the connector requires. The easiest application for connector development is one that interacts with its database through a high-level API. If the application provides a low-level API or has no API, the connector will probably require more coding.

Application use of batch clean-up or merge programs

A final aspect of the application business object interface that you need to investigate is whether the application uses any batch clean-up or merge programs to purge redundant or invalid data. For example, an application may run a batch program once a day to standardize site names that operators may have typed in incorrectly or incompletely. This program might, for example, change all sites named IBM WebSphere to IBM WebSphere Software.

When this type of batch program runs, all changes to the database may also need to flow through an InterChange Server customer synchronization system. A program like this may result in hidden requirements for your connector. For example, even if it appears initially that your connector does not need to provide Delete functionality, you may need to provide Delete functionality to support a batch clean-up program that deletes all sites named IBM WebSphere.

You may decide that you want to handle batch clean-up tasks periodically, such as once a month, rather than synchronously. In any case, an important planning task is to gather information about any programs that result in unanticipated requirements for your connector.

Designing application-specific business objects

Application-specific business objects are the units of work that are triggered within the application, created and processed by the connector, and sent to the integration broker. A connector uses these business objects to export data from its application to other applications and to import data from other applications.

The connector exposes all the information about an application entity that is necessary to allow other applications to share the data. Once the connector makes the entity available to other applications, the integration broker can route the data to any number of other applications through their connectors.

Designing the relationship between the connector and its supported application-specific business objects is one of the tasks in connector development. Application-specific business object design can generate requirements for connector programming logic that must be integrated into the connector development process. Therefore, business object and connector developers must work together to develop specifications for the connector and its business objects.

Consider the following design guidelines when you design your application-specific business objects:

1. Determine what application entities the connector will work with.
2. Determine the scope of business object development.
3. Determine support for a metadata-driven design.

Note: For more information about the design of application-specific business objects, see the *Business Object Development Guide*.

Determining the application entities

The complexity of business objects can have a significant impact on the amount of work that is necessary to build a connector. A first step in identifying application-specific business objects is to determine what application entities the connector will work with.

You can identify application entities that the connector will work with in two ways:

- Focus on existing InterChange Server collaborations whose business processes correspond to those of your application.
- Focus on other applications that you want to integrate with your application.

Design focus on InterChange Server collaborations

If you are using InterChange Server as your integration broker, one way to begin identifying application-specific business objects is to list the InterChange Server collaborations that you want the application to work with. Consider the features of each collaboration, and note which generic business objects each collaboration references. Using this list, you can decide what kinds of business objects allow your application to work with the collaboration.

For example, you may decide that you want to use your application with the Customer Manager collaboration. In this case, the connector must handle customer entities. The connector might extract customer data from the application to forward to the collaboration or receive customer data from the collaboration to pass back to the application.

Design focus on other applications

Alternatively, you might start the connector development task by looking at other applications with which you want to integrate. As you examine your application and other applications, you can determine what business processes you want to share across applications and identify what data you want to exchange. The goal is to determine what entities in your application make sense to implement as business objects to enable integration with other applications.

For example, if your application stores customer data, you may want to keep the customer database consistent with the customer database in another application. To synchronize customer data, you need to know about the customer entity that each application publishes. Figure 14 illustrates a design approach that focuses on integrating with other applications.

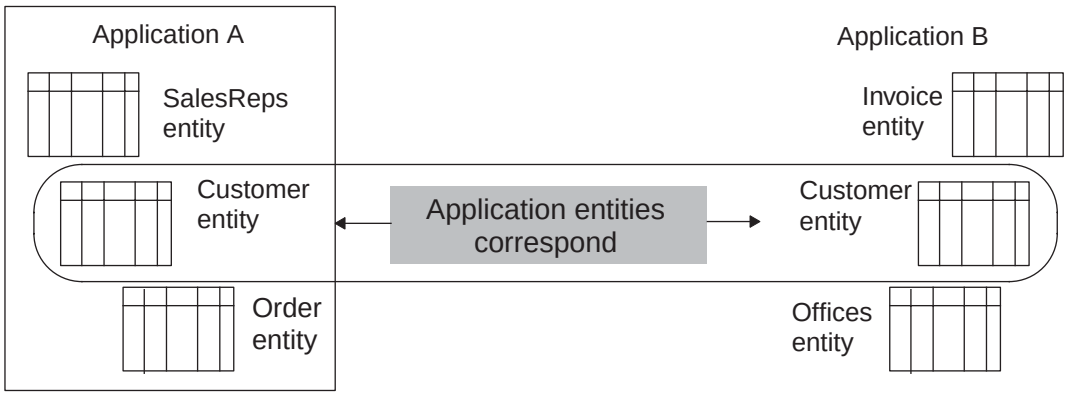

Figure 14. Design focus: identify applications with which to integrate

Design focus on the application

Use the following topics and questions to gather more information about application entities and business objects:

- "Contained entities"
- "Database representation of entities"
- "Denormalization of application entities" on page 46
- "Batch processing of application entities" on page 46

Contained entities:

- Do the application entities have contained entities?

  For example, in many applications a contract entity has one to many line items. The IBM WebSphere Business Integration Contract business object contains child line items as business objects. Determine whether the entities your connector will work with have related entities that will be defined as child business objects.

Database representation of entities:

- Are there application business entities that are the same type but that have different physical representations in the application?

  For example, an application may have two types of contracts: hardware contracts and software contracts. Both are of type Contract, but they are stored in different tables in the application database. In addition, the attributes for each Contract type differ.

  Because a single set of maps can convert between only one generic business object and one application-specific business object, developers for this application must design business objects to account for the different entities in the application. For example, they may need to redesign the IBM WebSphere Business Integration generic business object, create new generic child business objects, and create new maps.

  Figure 15 shows the business objects that may result from multiple application entities of the same type. It illustrates the creation of two generic child business objects, one that contains data specific to hardware contracts and one that contains data specific to software contracts.

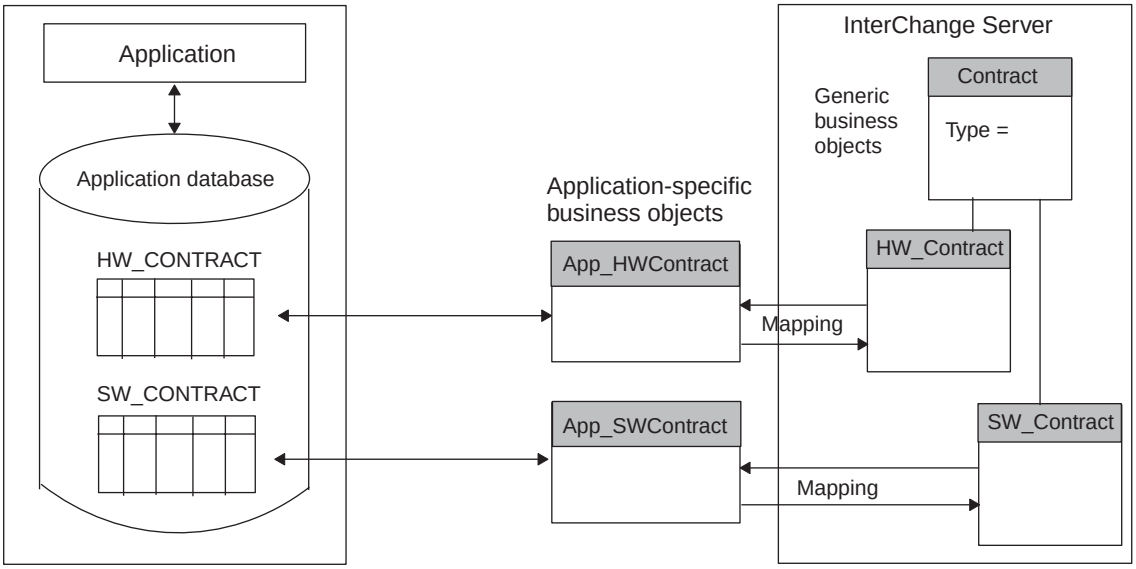

Figure 15. Database representation of application entities

Denormalization of application entities: Are there application entities that reside in more than one location in the database but that correspond to the same logical entity?

For example, Contract, Customer, and Contact entities might each have Customer address fields as part of the physical table definition for each entity. If the Customer address field changes in one entity, it must be updated in all entities.

However, the address fields might be consolidated into an Address business object that needs to be updated for the Contact, Customer, and Contract business objects if the address changes for any of the entities. In this case, the Address business object would be referenced rather than contained by the top-level business objects that use the data.

Batch processing of application entities: Are there batch processes associated with the creation of application entities?

In some applications, batch processing may add data to entities. As an example, a data entry operator may enter a new customer into the application database at 11:00 AM, but the customer record will not be complete until a 7:00 PM batch job runs to fill in some remaining values.

If a batch process is associated with application entities and the process adds important or required data, you need to determine when the business object is generated. For example:

- If the batch process generates the event notification, the event will trigger the connector to send a complete business object into the IBM WebSphere business integration system.
- If the operator's Save operation generates the event notification, the event may trigger the connector to send an incomplete business object.

If there is a need for real-time data synchronization, but there are batch processes running in the background, your connector development plans must account for this.

Determining the scope of business object development

When you have determined at a high level what business objects you need to define, you then need to determine the verb support for the business object development, as follows:

1. Use Table 13 to create a verb-scope summary for each business object and verb combination that your connector will support.
2. Use the completed scope summary to assemble information about each business object.

Table 13. Business Object Verb-Scoping Summary

Business object name	Required request Verbs (request processing)	Required delivery verbs (application event notification)
Object 1	O Create O Update O Delete	O Create O Update O Delete
Object 2	O Create O Update O Delete	O Create O Update O Delete
Object *n*	O Create O Update O Delete	O Create O Update O Delete

Important: Most connectors *must* support the Retrieve verb for each business object; therefore, it is not included in Table 13..

Assessing support for metadata-driven design

In addition to its structure and attributes, a business object definition can contain application-specific information, which can provide processing instructions or information on how the business object is represented in the application. Such information is called *metadata*.

Metadata can include any information that the connector needs in its interactions with the application. For example, if a business object definition for a table-based application includes metadata that provides the application table and column names, the connector can locate requested data using this information, and the application column names do not need to be encoded in the connector. Because the connector has access to its supported business object definitions at runtime, it can use the metadata in the business object definition to dynamically determine how to process a particular business object.

Depending on the application and its programming interface (API), a connector and its business objects might be designed based on the ability to support the use of metadata, as Table 14 shows.

Table 14. Connector support for metadata

Connector's use of metadata	Business object handlers required	For more information
Entirely driven by the processing instructions in the metadata of its business object definitions	One generic metadata-drive business object handler	"Metadata-driven connectors" on page 48
Partially driven by the metadata in its business object definitions	One partially metadata-driven business object handler	"Partially metadata-driven connectors" on page 49
Cannot use metadata	Separate business object handler for each business object that does not use metadata	"Connectors that do not use metadata" on page 50

While some application interfaces have constraints that restrict the use of metadata in connector and business object design, a worthwhile goal for connector

development is to make the connector as metadata driven as possible. Advantages and disadvantages of the approaches in Table 14 are discussed below.

Metadata-driven connectors

To be able to support metadata-driven design, the application API must be able to specify what objects in the application are to be acted upon. In general, this means that you can use the business object metadata to provide information about the application entity to be acted upon and the attribute data as the values for that object. A *metadata-driven connector* can then use the business object values and the metadata (the application-specific information that the business object definition contains) to build the appropriate application function calls or SQL statements to access the entity. The function calls perform the required changes in the application for the business object and verb the connector is processing.

Applications based on forms, tables, or objects are well suited for metadata-driven connectors. For example, applications that are forms-based consist of named forms. Programmatic interaction with a forms-based application consists of opening a form, reading or writing fields on the form, and then saving or dismissing the form. The connector for such an application can be driven directly by the business object definitions that the connector supports.

The main benefit to a metadata-driven connector is that the connector can use one generic business object handler for *all* business objects. In this approach, the business object definition contains *all* the information that the connector needs to process the business object. Because the business object itself contains the application-specific information, the connector can handle new or modified business objects without requiring modifications to the connector source code. The connector can be written in a generic manner, with a single *metadata-driven business object handler*, which does *not* contain hard-coded logic for processing specific business objects.

Note: Business object names should not have semantic value to the connector. The connector should process identically two business objects with the same structure, data, and application-specific information with different names.

WebSphere InterChange Server

Figure 16 shows an application-specific business object and a connector with a meta-data-driven business object handler. The processing instructions in the application-specific information of the `App_Order` business object tell the connector how to process the business object.

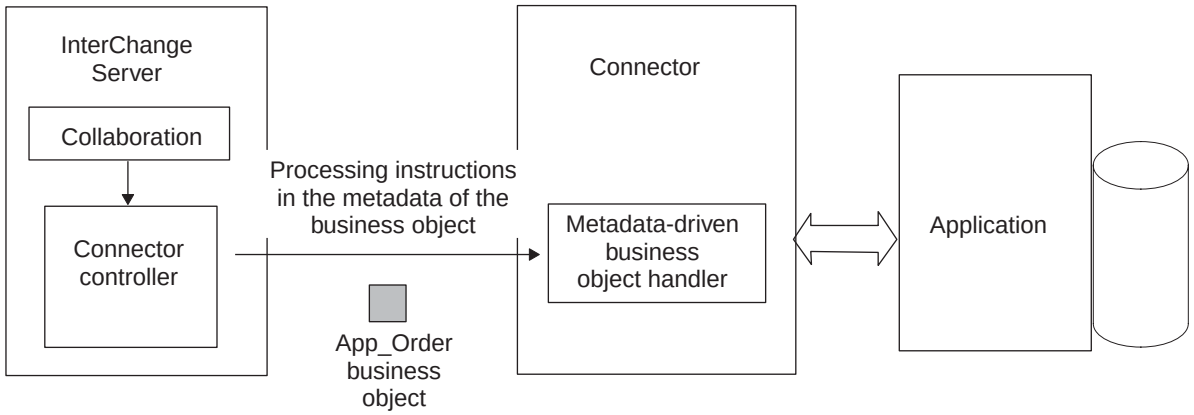

Figure 16. Using metadata in the business object for processing instructions

Because a metadata-driven connector derives its processing instructions from its application-specific business objects, the business objects must be designed with this type of processing in mind. This approach to connector and business object design provides flexibility and easy extensibility, but it requires more planning in the design phase. When connectors are designed to work with business object metadata, the business object itself can be changed without requiring corresponding changes in the connector.

For more information on designing a metadata-driven business object handler, see "Implementing metadata-driven business object handlers" on page 80.

Partially metadata-driven connectors

IBM encourages the metadata approach for designing connectors and application-specific business object definitions. However, some applications might not be suited for this approach. Application APIs that are specific for each entity in an application make it more difficult to write a metadata-driven connector. Often the issue is that the call itself differs between objects in some structural way, rather than just in the name of the method or the data that is passed.

Sometimes you can still drive a connector with metadata, though this metadata does not contain the actual processing instructions. This *partially metadata-driven connector* can use the metadata in the business object definition or attributes to help determine what processing to perform. For example, an application that has a large amount of business logic embedded in its user interface might have restrictions on how an external program, such as a connector, can get information into and out of its database. In some cases, it may be necessary to provide an extension to the application using the application environment and application programming interface. You may need to add object-specific modules to the application to handle the processing for each business object. The application may require the use of its application environment and interface to ensure that application business logic is enforced and not bypassed.

In this case, the business object and attribute application-specific information can still contain metadata for the connector. This metadata specifies the name of the module or API call needed to perform operations for the business object in the application. The connector can still be implemented with a single business object handler, but it is a *partially metadata-driven business object handler* because this metadata does not contain the processing instructions.

Figure 17 illustrates an application extension that is responsible for handling requests from the connector. The extension contains separate modules for each business object supported by the connector.

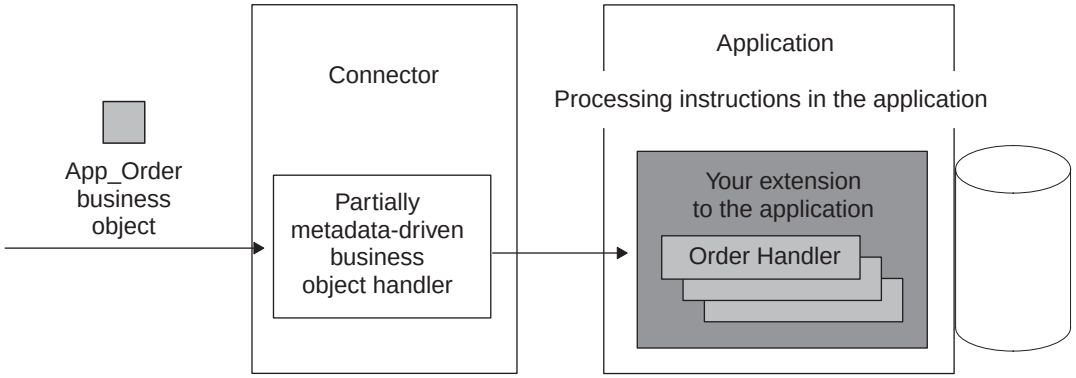

Figure 17. Application-specific processing in the application

The benefit to the partially metadata-driven connector is that it still uses just one business object handler. However, unlike with a metadata-driven connector, there is coding to do when new business objects are created for the connector. In this case, new object functions must be written and added to the application, but the connector does not need to be recoded or recompiled.

Connectors that do not use metadata

If the application API does *not* provide the ability to specify what entities in the application are to be acted upon, the connector cannot use metadata to support a single business object handler. Instead, it must provide *multiple business object handlers*, one for each business object the connector supports. In this approach, each business object handler contains specific logic and code to process a particular business object.

In Figure 18,, the connector has multiple, object-specific business object handlers. When the connector receives a business object, it calls the appropriate business object handler for that business object.

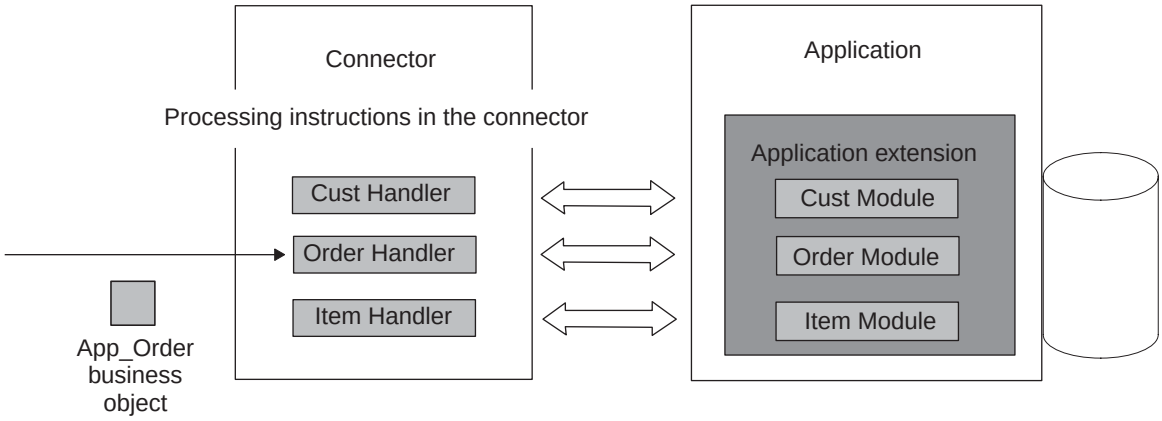

Figure 18. Application-specific processing in the connector

The drawback of this non-metadata approach is that when a business object is changed or a new business object is added, this type of connector must be recoded to handle the new or changed business object.

Event notification

The IBM WebSphere business integration system is an event-driven system, and connectors need some way to detect and record events that occur in the application. When you examine the application, determine whether it provides an event-notification mechanism that can notify the connector of changes to application data.

Event notification typically consists of a collection of processes that allows a connector to be notified of internal application events. The event record should include the type of the event, the business object name and verb, such as Customer and Create, and the data key required for the connector to retrieve associated data.

In addition, an event-notification strategy must incorporate the necessary mechanisms to ensure the data integrity between event records and the corresponding event data. In other words, an event notification should not occur until *all* the required data transactions for the event have completed successfully.

The design of an event notification mechanism varies depending on the extent to which the application reports application events and enables clients to retrieve event data. If the application provides an event notification interface such as an API, IBM recommends that you use this to implement the event-notification mechanism. The use of an API helps ensure that connector interactions with the application abide by application business logic. If the application provides an event-notification mechanism, use the following topics and questions to gather more information.

Event notification level of detail

- Does the application's event-notification mechanism provide enough detail about the event to establish the discrete business object and verb? If not, can the event notification component be configured to provide this level of detail?

  For example, if a new record is added or an existing customer is updated, determine whether the event-notification mechanism can provide information on the type of operation, such as Create or Update operations. If the connector supports delta operations, determine whether the event mechanism can provide information on exactly which subobjects or attributes changed.

Event notification support for business logic

- Does event notification occur at a level that adequately supports business requirements? In other words, an event-notification mechanism would ideally include support for application business logic.

In your project plan, describe the event-notification mechanism. If there is no existing event mechanism, determine what alternatives are available to detect changes to application data. For example, you might be able to provide event notification by setting up database triggers on tables in a relational database. Or the application might provide a batch-export capability that exports all database modifications to a file from which the connector can extract information about application events.

Note: For more information on the stages of implementing an event-notification mechanism, see "Overview of an event-notification mechanism" on page 115.

Communication across operating systems

Communication between the application and the connector is a major component in the overall connector design. If the application runs on a different operating system from InterChange Server and the connector, you must ensure that a mechanism is in place to allow the connector access to the application.

If the application provides an API, determine whether the API handles the communication between the operating system of application and that of the connector. For example, if the application runs on UNIX and the connector and InterChange Server run on Windows 2000, the application API might enable the connector and application to communicate across operating systems.

Figure 19 shows an example communication mechanism between an ODBC connector running on Windows 2000 and an ODBC-based application running on UNIX. The connector builds dynamic SQL statements and executes them using the ODBC API. The ODBC driver enables the connector to establish a connection with the application database and to access the database using ODBC SQL statements.

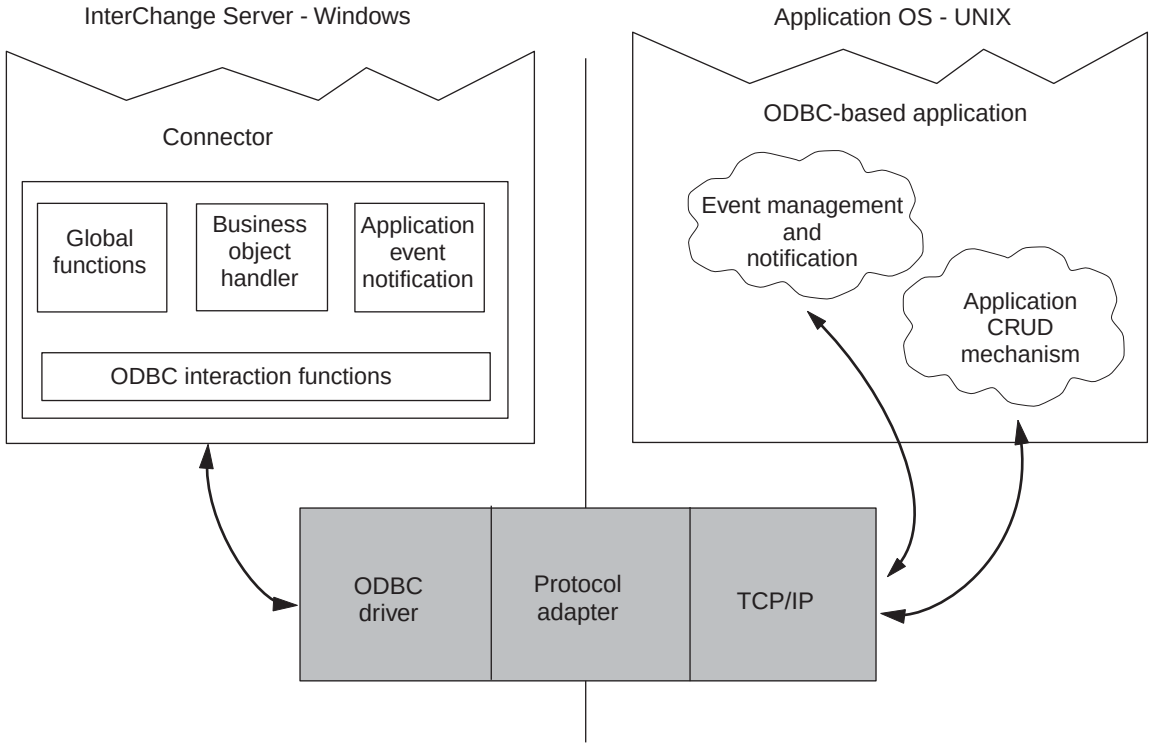

Figure 19. Sample Windows-to-UNIX communication

Summary set of planning questions

The following table lists the set of planning questions provided in this chapter. You can use this table as a worksheet for gathering information about your application. As you gather information, get copies of any documentation that can help in the planning, design, or development phases of the project.

1. Understanding the Application
   - What is the application operating system?
   - What programming languages were used to create the application?
   - What is the execution architecture of the application?
   - Is there a central database for application data? What type of database is it?
   - Is the application or its database distributed across multiple servers?
2. Identifying the Directionality of the Connector
   - Does the connector need to send data, receive data, or both?
3. Identifying the Application-Specific Business Objects
   - Do application entities have contained entities?
   - Are there application business entities that are the same type but have different physical representations in the application?
   - Are there application entities that reside in more than one location in the database but correspond to the same logical entity?
   - Are there batch processes associated with the creation of application entities?
4. Investigating the Application Data Interaction Interface
   - Have there been any other efforts to integrate with this application?
     - What was the purpose of the integration?
     - Does the integration use interfaces that modify or retrieve information?
     - If the integration is able to process an event generated in the application, what is the mechanism used to trigger event processing?
     - Will your connector replace the pre-existing integration?
   - Is application data shared by other applications?
     - Do other applications create, retrieve, update, or delete this application's data?
     - What is the mechanism used by other applications to gain access to the data?
     - Is there object-specific business logic used by other applications?
   - Is there a mechanism that the connector can use to communicate with the application?
     - Does the API allow access for create, retrieve, update, and delete operations?
     - Does the API provide access to all data entity attributes?
     - Does the API allow access to the application for event detection?
     - Are there inconsistencies in the API implementation?
     - Describe the transaction behavior of the API.
     - Is the API suited for meta-data design?
     - Does the API enforce application business rules?
   - Are there batch clean-up or merge programs used to purge redundant or invalid data?
5. Investigating the Event Management and Notification Mechanism
   - Describe the event management mechanism.
   - Does it provide the necessary granularity to establish the distinct object and verb?
   - Does event notification occur at a level that can support application business logic?
6. Investigating Communication Across Operating Systems
   - Does the API handle the communication mechanism between the application operating system and the connector operating system?
   - If not, is there a mechanism available to handle communication across operating systems?

Figure 20. Summary set of planning question

Evaluating the findings

As you assemble the answers to the questions presented in this chapter, you acquire essential information about application data entities, business object processing, and event management. These findings become the basis for a high-level architecture for the connector.

When you have determined what entities your connector will support and have examined the application functionality for database interaction and event notification, you should have a clear understanding of the scope of the connector development project. At this point, you can continue with the next phases of connector development—defining application-specific business objects and coding the connector.

Figure 21 shows a partial write-up of information about a sample connector. Figure 22 illustrates a high-level architecture diagram for an ODBC-based connector.

1. Understanding the Application
   - Application is running on UNIX.
   - Programming language used is Visual C++ with the Microsoft MFC libraries.
   - Application is client-server.
   - Application has a central database. Type is RDMS.
   - Application is not distributed.
2. Identifying the Directionality of the Connector
   - Connector will be bidirectional.
3. Identifying the Application-Specific Business Objects
   - Business objects have contained objects. Contained business objects are:
     - Customer ″Address ″Site Use and Site Profile
     - Item ″Status
     - Contact ″*n* Phones and *n* Roles
   - Application business entities do not have different physical representations in the application.
   - Application entities do not reside in more than one location in the database.
   - No batch processes are associated with the creation of these objects.
4. Examining the Application Data Interaction Interface
   - No previous efforts to integrate with this application.
   - Application data is not shared by other applications.
   - The application provides the OpenProduct API.
     - OpenProduct allows for Creates and Updates but not Retrieves and Deletes.
     - The API provide access to all data entity attributes.
     - The API allows access to the application for event detection. We can create an event table and poll for events at a specified interval.
     - There are no inconsistencies in the API.
     - The API has a batch interface.
     - The application is table-based, and the API is suited for meta-data design.
     - ...

Figure 21. Sample results write-up

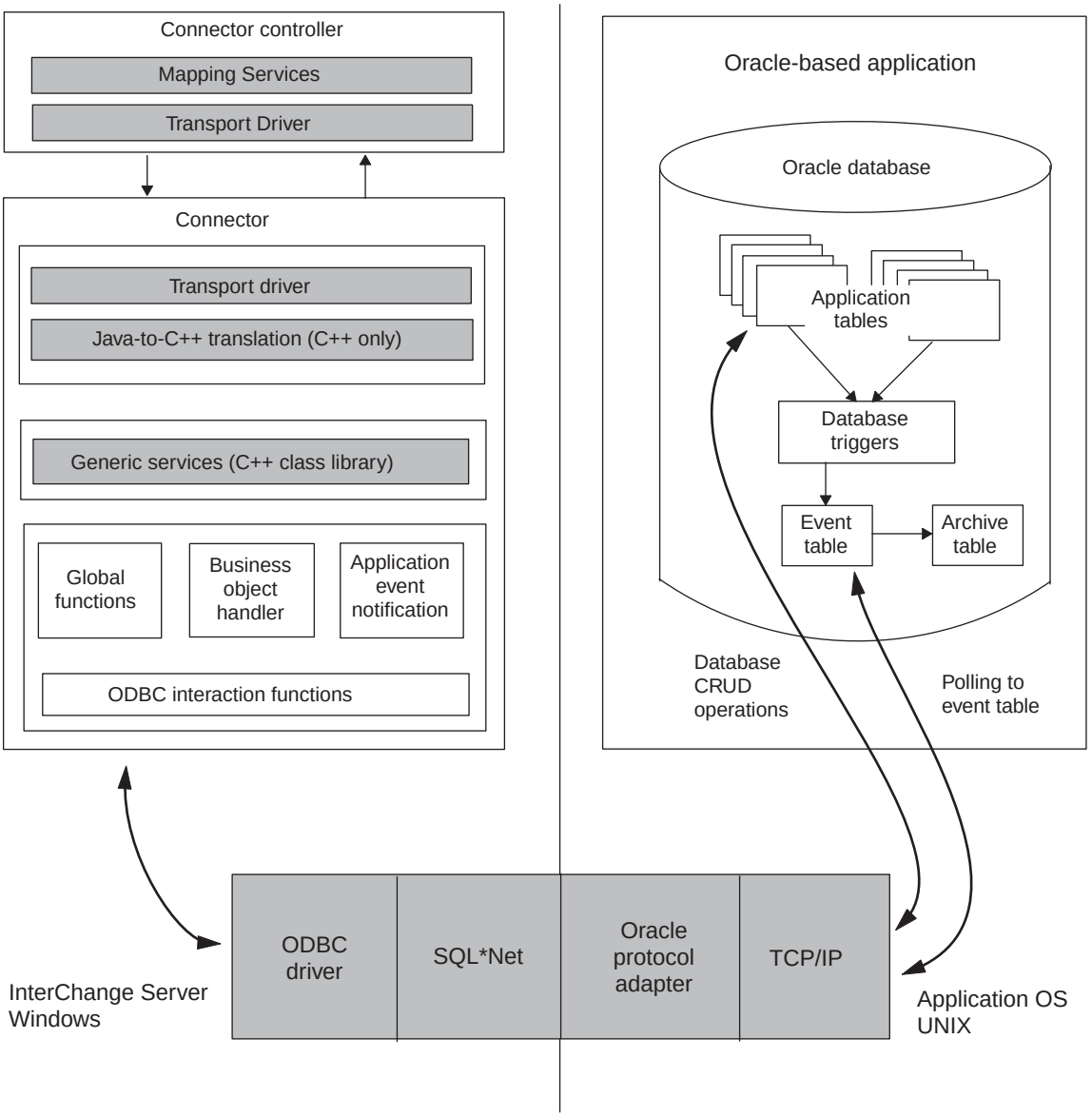

Figure 22. Sample ODBC-based connector architecture

An internationalized connector

An *internationalized connector* is a connector that has been written so that it can be customized for a particular locale. A *locale* is the part of a user's environment that brings together information about how to handle data that is specific to the end user's particular country, language, or territory. The locale is typically installed as part of the operating system. Creating a connector that handles locale-sensitive data is called the *internationalization* (I18N) of the connector. Preparing an internationalized connector for a particular locale is called the *localization* (L10N) of the connector.

This section provides the following information on an internationalized connector:

- "What is a locale?" on page 56
- "Design considerations for an internationalized connector" on page 56

What is a locale?

A *locale* provides the following information for the user environment:

- Cultural conventions according to the language and country (or territory):
  - Data formats:
    - Dates: define full and abbreviated names for weekdays and months, as well as the structure of the date (including date separator).
    - Numbers: define symbols for the thousands separator and decimal point, as well as where these symbols are placed within the number.
    - Times: define indicators for 12-hour time (such AM and PM indicators) as well as the structure of the time.
    - Monetary values: define numeric and currency symbols, as well as where these symbols are placed within the monetary value.
  - Collation order: how to sort data for the particular character code set and language.
  - String handling includes tasks such as letter "case" (upper case and lower case) comparison, substrings, and concatenation.
- A *character encoding* — the mapping from a character (a letter of the alphabet) to a numeric value in a character code set. For example, the ASCII character code set encodes the letter "A" as 65, while the EBCIDIC character set encodes this letter as 43. The *character code set* contains encodings for all characters in one or more language alphabets.

A locale name has the following format:

ll_TT.codeset

where *ll* is a two-character language code (usually in lower case), *TT* is a two-letter country and territory code (usually in upper case), and *codeset* is the name of the associated character code set. The *codeset* portion of the name is often optional. The locale is typically installed as part of the installation of the operating system.

Design considerations for an internationalized connector

This section provides the following categories of design considerations for internationalizing a connector:

- "Locale-sensitive design principles"
- "Character-encoding design principles" on page 60

Locale-sensitive design principles

To be internationalized, a connector must be coded to be locale-sensitive; that is, its behavior must take the locale setting into consideration and perform the task appropriate to that locale. For example, for locales that use English, the connector should obtain its error messages from an English-language message file. The WebSphere Business Integration Adapters product provides you with an internationalized version of the connector framework. To complete the internationalization (I18N) of a connector you develop, you must ensure that your application-specific component is internationalized.

Table 15 lists the locale-sensitive design principles that an internationalized application-specific component must follow.

Table 15. Locale-sensitive design principles for application-specific components

Design principle	For more information
The text of all error, status, and trace messages should be isolated from the application-specific component in a message file and translated into the language of the locale.	"Text strings"
The locale of a business object must be preserved during execution of the connector.	"Business object locales" on page 58
Properties of connector configuration properties must be handled to include possible inclusion of multibyte characters.	"Connector configuration properties" on page 59
Other locale-specific tasks must be considered.	"Other locale-sensitive tasks" on page 59

Text strings: It is good programming practice to design a connector so that it refers to an external message file when it needs to obtain text strings rather than hardcoding text strings in the connector code. When a connector needs to generate a text message, it retrieves the appropriate message by its message number from the message file. Once all messages are gathered in a single message file, this file can be localized by having the text translated into the appropriate language or languages.

This section provides the following information on how to internationalize text strings:

- "Handling logging and tracing"
- "Handling miscellaneous strings" on page 58

Handling logging and tracing: To internationalize the logging and tracing, make sure that all these operations use message files to generate text messages. By putting message strings in a message file, you assign a unique identifier to each message. Table 16 lists the types of operations that use a message file and the associated Java connector library methods in the `CWConnectorUtil` class that the application-specific component uses to retrieve their messages from a message file.

Table 16. Methods to log and trace messages from a message file

Message-file operation	Connector library method
Logging	`generateAndLogMsg()`
Tracing	`generateAndTraceMsg()` or `traceWrite()`

Log messages should display in the language of the customer's locale. Therefore, log messages should always be isolated into a connector message file and retrieved with the `generateAndLogMsg()` method.

Because trace messages are intended for the product debugging process, they often do not need to display in the language of the customer's locale. Therefore, whether trace messages are contained in a message file is left at the discretion of the developer:

- If non-English-speaking users need to view trace messages, you need to internationalize these messages. Therefore, you must put the trace messages in a message file and extract them with the `generateMsg()` method. This message file should be the connector message file, which contains message specific to your

connector. The `generateMsg()` method generates the message string for `traceWrite()`. It retrieves a predefined trace message from a message file, formats the text, and returns a generated message string.

- If only English-speaking users need to view trace messages, you do not need to internationalize these messages. Therefore, you can include the trace message (in English) directly in the call to `traceWrite()`. You do *not* need to use the `generateMsg()` method.

  However, storing trace messages in the message file makes it easy to locate and maintain them.

Handling miscellaneous strings: In addition to handling the message-file operations in Table 16,, an internationalized connector must *not* contain any miscellaneous hardcoded strings. You must isolate these strings into the message file as well. Table 17 shows the method that the application-specific component can use to retrieve a message from a message file.

Table 17. Method to retrieve a message from the message file

Connector library class	Method
`CWConnectorUtil`	`generateMsg()`

To internationalize hardcoded strings, take the following steps:

- Generate a uniquely numbered message in the connector message file for the hardcoded string.

  Note: In the message file, you can also include an optional explanation to the isolated string. In this explanation, you can put the method name where the string is used. This information can help to track the position of the source and make changes when needed.

- In the application-specific component, use the `generateMsg()` method to specify the isolated string by its message number.

For example, suppose your application-specific component contains the following hardcoded string in a line of code:

```
********Before updating the event status********
```

To isolate this hardcoded string from the connector code, create a message in the message file and assign it a unique message number (100):

```
100
********Before updating the event status********
[EXPL]
Hardcoded message in pollForEvents()
```

The application-specific component retrieves the isolated string (message 100) from the message file and replaces the hardcoded string with this retrieved string:

```
//retrieve the message numbered '    100'
String msg100 = generateMsg(100, CWConnectorLogAndTrace.XRD_INFO,
    CWConnectorLogAndTrace.CONNECTOR_MESSAGE_FILE, 0);
MyClassObject.formatMsg(msg100);  //send retrieved message to a custom method
```

For more information on the use of message files, see Chapter 6, “Message logging,” on page 139.

Business object locales: The connector might need to perform locale-sensitive processing (such as data format conversions) when it converts from application

data to the application-specific business object. During processing of a business object in a connector, there are two different locale settings:

- The connector inherits a locale, called the *connector-framework locale*, from the connector framework with which it runs. The connector-framework locale determines the locale of text messages that the connector uses for logging and exceptions.
- The connector also can access the locale that is associated with a business object it is processing. This *business-object locale* identifies the locale associated with the data in the business object.

Table 18 shows the method that the connector can use to retrieve the locale associated with the connector framework.

Table 18. Method to retrieve the connector framework's locale

Connector Library Class	Method
`CWConnectorUtil`	`getGlobalLocale()`

When a business object is created, it can have a locale associated with its data. Your connector can access this business-object locale in either of the following ways:

- To obtain the name of the business-object locale, use the `getLocale()` method, which is defined in the `CWConnectorBusObj` class. The `CWConnectorBusObj` class also provides a `setLocale()` method.
- To associate a locale with the business object, use the `createBusObj()` method, which is defined in the `CWConnectorUtil` class.

Connector configuration properties: As discussed in "Using connector configuration property values" on page 70,, an application-specific component can use two types of configuration properties to customize its execution:

- Standard configuration properties are available to all connectors.
- Connector-specific configuration properties are unique to the particular connector in which they are defined.

The names of all connector configuration properties must use *only* characters defined in the code set associated with the U.S English (`en_US`) locale. However, the values of these configuration properties can contain characters from the code set associated with the connector framework locale.

The application-specific component obtains the values of configuration properties with the methods described in "Retrieving connector configuration properties" on page 71.. These methods correctly handle characters from multibyte code sets. However, to ensure that your connector is internationalized, its code must correctly handle these configuration-property values once it retrieves them. The application-specific component must *not* assume that configuration-property values contain only single-byte characters.

Other locale-sensitive tasks: An internationalized connector must also handle the following locale-sensitive tasks:

- Sorting or collation of data: the collaboration must use a collation order appropriate for the language and country of the locale.
- String processing (such as comparison, substrings, and letter case): the collaboration must ensure that any processing it performs is appropriate for characters in the locale's language.

- Formats of dates, numbers, and times: the collaboration must ensure that any formatting it performs is appropriate for the locale.

Character-encoding design principles

If data transfers from a location that uses one code set to a location that uses a different code set, some form of character conversion needs to be performed for the data to retain its meaning. The Java runtime environment within the Java Virtual Machine (JVM) represents data in Unicode. The Unicode character set is a universal character set that contains encodings for characters in most known character code sets (both single-byte and multibyte). There are several encoding formats of Unicode. The following encodings are used most frequently within the integration business system:

- Universal multiple octet Coded Character Set: UCS-2

  The UCS-2 encoding is the Unicode character set encoded in 2 bytes (octets).
- UCS Transformation Format, 8-bit form: UTF-8

  The UTF-8 encoding is designed to address the use of Unicode character data in UNIX environments. It supports all ASCII code values (0...127) so that they are never interpreted as anything except a true ASCII code. Each code value is usually represented as a 1-, 2-, or 3- byte value.

Most components in the WebSphere business integration system, including the connector framework, are written in Java. Therefore, when data is transferred between most system components, it is encoded in the Unicode code set and there is no need for character conversion.

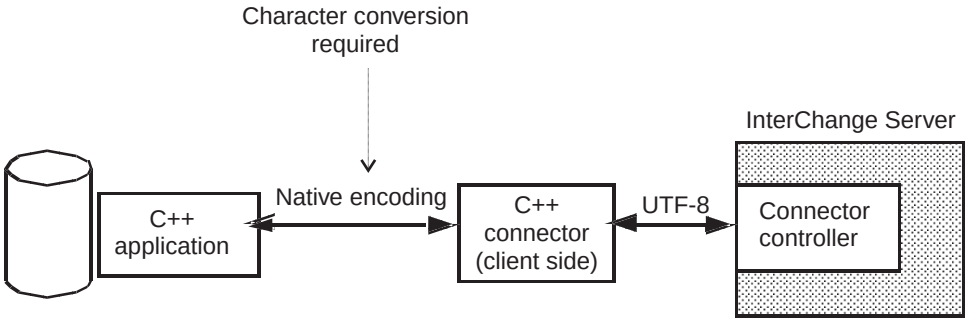

Figure 23. Character encoding with a C++ connector

Because a Java connector works with a Java application (or technology), its application-specific component is written in Java, which handles data in the Unicode code set. The Java application (or technology) also has data already in Unicode. Therefore, a Java connector does not normally need to perform character conversion on application data for the application-specific business object. If some data is not in Unicode, the Java environment automatically supports character conversion between UCS-2 and a native encoding. However, if the application data includes data from some other external source (such as an operating-system file), the Java connector might need to handle character conversion. Figure 24 shows the character encoding for a Java connector.

Note: A connector obtains the character encoding of its application from the `CharacterEncoding` connector configuration property. If your connector performs character conversion, make sure you instruct the connector end user to set this connector property to the correct value.

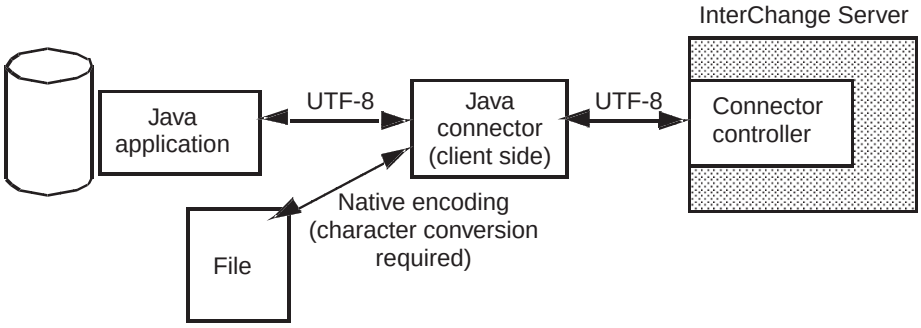

Figure 24. Character encoding with a Java connector

To obtain the character encoding at runtime, Table 19 shows the method in the Java connector library that the connector can use.

Table 19. Method to retrieve the connector framework's character encoding

Connector library class	Method
`CWConnectorUtil`	`getGlobalEncoding()`

A Java `String` is UCS-2 encoded. Therefore, the connector can get and set attribute values (those represented as Java `String`s), default attribute values, and application-specific information in their native encoding by performing a simple conversion:

```
nativeEncodedAppSpecInfo = busObj.getAppText(attrName).getBytes(nativeEncoding);
```

Note: Connector configuration properties with String values do not require character conversion because they originate from the InterChange Server repository and are therefore in the UCS-2 encoding.

Chapter 3. Providing general connector functionality

This chapter presents information on how to implement a *connector class*, which performs the initialization and setup for the application-specific component of a connector. It also discusses some basic functionality that your connector might need.

Note: Writing code for the application-specific component is only one part of the overall task for developing a connector. Before you begin to write your application-specific component, you should clearly understand the connector design issues as well as the design of any application-specific business objects. A thorough understanding of the design issues can help you complete the coding task successfully. For information on connector design, refer to Chapter 2, "Designing a connector," on page 37.

This chapter contains the following sections:

Running a connector

When the connector runs, it performs the tasks summarized in Table 20..

Table 20. Steps for executing a connector

Execution step		For more information
1.	Start the connector with the startup script to initialize the connector framework and application-specific component of the connector.	"Starting up a connector" on page 63
2.	If polling is turned on, the connector framework calls `pollForEvents()` at the interval defined by the connector's `PollFrequency` connector configuration property.	"Polling for events" on page 67
3.	If the connector implements request processing, call the business-object handler associated with the request business object that the connector receives.	Request processing is implemented by the `doVerbFor()` method in the connector's business object handler. For more information, see Chapter 4, "Request processing," on page 79.
4.	When the connector is shut down, the connector framework calls `terminate()`.	"Shutting down the connector" on page 68

The following sections provide more information about each of the execution steps Table 20..

Starting up a connector

Each connector has a connector startup script to begin its execution. This startup script invokes the connector framework.

Note: For more information on how to create a connector startup script, see "Creating startup scripts" on page 215..

Once the connector framework is executing, it performs the appropriate steps to invoke the application-specific component of the connector, based on the integration broker.

Starting connectors with InterChange Server

When InterChange Server is the integration broker, the connector framework takes the following steps to invoke the application-specific component:

1. Use the Object Request Broker (ORB) to establish contact with InterChange Server.
2. From the repository, load the following connector-definition information into memory for the connector's process:
   - The connector configuration properties
   - A list of the connector's supported business object definitions
3. Begin execution of the connector's application-specific component by instantiating the connector base class and calling methods of this base class that initialize the application-specific component.

   When the connector is started, the connector framework instantiates the connector base class and then calls the connector-base-class methods in Table 21..

Table 21. Beginning execution of the connector

Initialization task		For more information
1.	Initialize the connector to perform any necessary initialization for the application-specific component, such as opening a connection to the application.	"Initializing the connector" on page 65
2.	For each business object that the connector supports, obtain the business object handler.	"Obtaining the business object handler" on page 66

   Once these methods have been called, the connector is operational.
4. Contact the connector controller to obtain the subscription list for business objects to which collaborations have subscribed. For more information, see "Business object subscription and publishing" on page 13..

Starting connectors with other integration brokers

When a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server is the integration broker, the connector framework takes the following steps to invoke the application-specific component:

1. From the local repository, load the following connector-definition information into memory for the connector's process:
   - The connector configuration properties
   - A list of the connector's supported business object definitions
2. Begin execution of the connector's application-specific component by instantiating the connector base class and calling methods of this base class that initialize the application-specific component.

   When the connector is started, the connector framework instantiates the connector base class and then calls the connector-base-class methods in Table 21.. Once these methods have been called, the connector is operational.

Initializing the connector

To begin connector initialization, the connector framework calls the initialization method of the connector base class. Table 22 shows the initialization method for the connector.

Table 22. Connector Base Class Methods to Initialize the Connector

Class	Method
`CWConnectorAgent`	`agentInit()`

As part of the implementation of the connector class, you *must* implement an initialization method for your connector. The main tasks of the initialization method include:

- "Establishing a connection"
- "Checking the connector version"
- "Recovering In-Progress events"

Important: During execution of the initialization method, business object definitions and the connector framework's subscription list are *not* yet available.

Establishing a connection: The main task of the initialization method is to establish a connection to the application. To establish the connection, the initialization method can perform the following tasks:

- Read from the repository the connector's configuration properties that provide connector information (such as `ApplicationUserID` and `ApplicationPassword`) and use them to send login information to the application. If a required connector property is empty, your initialization method can provide a default value.

  Use the `getConfigProp()` method to obtain the value of a connector configuration property. For more information, see "Using connector configuration property values" on page 70.
- Obtain any required connections or files. For example, the initialization method usually opens a connection with the application. It returns "success" if the connector succeeds in opening a connection. If the connector *cannot* open a connection, the initialization method must return the appropriate failure status to indicate the cause of the failure.

  In a Java connector, the `agentInit()` method should throw the `ConnectionFailureException` exception if the connection fails or the `LogonFailureException` exception if the connector is unable to log into the application. For information on these conditions, see "Exceptions" on page 204.

Checking the connector version: The `getVersion()` method returns the version of the connector. It is called in both of the following contexts:

- The initialization method should call `getVersion()` to check the connector version.
- The connector framework calls the `getVersion()` method when it needs to get a version for the connector.

Note: A connector should keep track of which application versions it supports. It should check the application version when it logs on to the application.

Recovering In-Progress events: Processing an event during event notification includes performing a retrieve on the application entity, creating a new business

object for the event, and sending the business object to the connector framework. If the connector terminates while processing an event and before updating the event status to indicate that the event was either sent or failed, the In-Progress event will remain in the event store. When a connector is restarted, it should check the event store for events that have an In-Progress status.

If the connector finds events with the In-Progress status, it can choose to do one of the tasks outlined in Table 23.. This behavior should be configurable. Several connectors use the `InDoubtEvents` connector configuration property for this purpose. Its settings are also shown in Table 23..

Table 23. Actions to take to recover In-Progress events

Event-recovery action taken	Value of InDoubtEvents
Change the status of the In-Progress events to Ready-for-Poll so they can be submitted to the connector framework in subsequent poll calls. **Note:** If events are resubmitted, duplicate events might be generated. If you want to ensure that duplicate events are not generated during recovery, use another recovery response.	`Reprocess`
Log a fatal error, shutting down the connector. If `LogAtInterchangeEnd` is set to True, this triggers an email notification about the error.	`FailOnStartup`
Log an error without shutting down the connector.	`LogError`
Ignore the In-Progress event records in the event store.	`Ignore`

For a Java connector, the `CWConnectorEventStore` class provides the `recoverInProgressEvents()` method to obtain event records with an In-Progress status from the event store and take the appropriate recovery action. The connector developer can implement this method to take actions based on the value of `InDoubtEvents`. In t his method, the connector developer can also change the status of in-progress events to the ready-for-poll status.

Note: For more information on event notification, the event store, and In-Progress events, see Chapter 5, "Event notification," on page 115.

Obtaining the business object handler

As the final step in connector initialization, the connector framework obtains the business object handler for each business object definition that the connector supports. A *business object handler* receives request business objects from the connector framework and performs the verb operations defined in these business objects. Each connector must have a `getConnectorBOHandlerForBO()` method defined in its connector base class to retrieve the business object handler. This method returns a reference to the business object handler for a specified business object definition.

Important: As part of the implementation of the connector base class, you *must* implement the `getConnectorBOHandlerForBO()` to obtain business object handlers for your connector.

To instantiate the business object handler (or business object handlers), the connector framework takes the following steps:

1. During initialization, the connector framework receives a list of business object definitions that the connector supports. For more information, see "Starting up a connector" on page 63.

2. The connector framework then calls the `getConnectorBOHandlerForBO()` method, once for every supported business object.
3. The `getConnectorBOHandlerForBO()` method instantiates the appropriate business object handler for that business object, based on the name of the business object definition it receives as an argument. It returns the business object handler to the connector framework.

   The number of business object handlers that are instantiated depends on the overall design of your connector's business object handling:

   - If the business object definitions for application-specific business objects contain metadata that follows consistent rules, the connector is metadata-driven. It can be designed to use a *metadata-driven business object handler*.

     A metadata-driven connector handles *all* business objects in a single, generic business object handler, called a metadata-driven business object handler. Therefore, the `getConnectorBOHandlerForBO()` method can simply instantiate one business object handler, regardless of the number of business objects the connector supports. It can create a business object handler the first time it is called and return a pointer to the same handler for each subsequent call.
   - If some or all application-specific business objects require special processing, then you must set up *multiple business object handlers* for those objects.

     If your connector requires a separate business object handler for each business object, the `getConnectorBOHandlerForBO()` method can instantiate the appropriate business object handler, based on the name of the business object being passed in. In this case, `getConnectorBOHandlerForBO()` instantiates multiple business object handlers, one for each business object definition that requires a separate business object handler. Each time the business-object-handler retrieval method is called, it instantiates a separate business object handler.
4. The connector framework stores the reference to this business object handler in the associated business object definition (which resides in the memory of the connector's process).

Important: Before you implement the `getConnectorBOHandlerForBO()` method, you want to complete the design for business object handling for your connector. For information on designing application-specific business object, see "Assessing support for metadata-driven design" on page 47..

For more information on how to implement the `getConnectorBOHandlerForBO()`method, see "Obtaining the Java business object handler" on page 153. For information on how to implement business object handlers, see Chapter 4, "Request processing," on page 79.

Polling for events

If a connector is to implement event notification, it must implement an event notification mechanism. Event notification contains methods that interact with an application to detect changes to application business entities. These changes are represented as events, which the connector sends to the connector framework for routing to a destination (such as InterChange Server).

If the connector uses a polling mechanism for event notification, the connector must implement the `pollForEvents()` method to periodically to retrieve event information from the event store, which holds events that the application generates until the connector can process them. When polling is turned on, the connector

framework calls the poll method `pollForEvents()`. The `pollForEvents()` method returns an integer indicating the status of the polling operation.

In the Java connector library, the `pollForEvents()` method is defined in the `CWConnectorAgent` class. Typical return codes used in `pollForEvents()` are `SUCCEED`, `FAIL`, and `APPRESPONSETIMEOUT`. For more information on return codes, see "Java return codes" on page 203.

Important: The developer must provide an implementation of the `pollForEvents()` method. If the connector supports *only* request processing, you do not need to fully implement `pollForEvents()`. However, because the poll method is a required method, you must implement a stub for the method. The Java connector library provides a default implementation of the `pollForEvents()` method.

For a more thorough discussion of event notification and the implementation of `pollForEvents()`, see Chapter 5, "Event notification," on page 115.

Shutting down the connector

The administrator shuts down a connector with by terminating the connector startup script. When the connector is shut down, the connector framework calls the `terminate()` method of the connector base class. The main task of the `terminate()` method is to close the connection with the application and to free any allocated resources.

Extending the connector base class

To create a connector, you extend the *connector base class*, available in the connector library. The base class for the connector includes methods for initialization and setup of the connector's application-specific component. Your derived *connector class* contains the code for the application-specific component of the connector.

Note: For information on naming conventions for a connector, see *Naming IBM WebSphere InterChange Server Components* in the IBM WebSphere InterChange Server documentation set.

The connector base class includes the methods shown in Table 24.. You must implement these methods in your connector.

Table 24. Methods to implement in the connector base class

Description	Connector base class method	For more information
Initializes the connector's application-specific component	`agentInit()`	"Initializing the connector" on page 65
Returns the version of the connector	`getVersion()`	"Checking the connector version" on page 65
Sets up one or more business object handlers	`getConnectorBOHandlerForBO()`	"Obtaining the business object handler" on page 66
Polls for application events	`pollForEvents()`	"Polling for events" on page 67
Performs cleanup tasks upon connector termination	`terminate()`	"Shutting down the connector" on page 68

Figure 25 illustrates the complete set of methods that the connector framework calls, and shows which methods are called at startup and which are called at runtime. All but one of the methods that the connector framework calls are in the

connector base class. The remaining method, `doVerbFor()`, is in the business object handler class; for information on implementing the `doVerbFor()` method, see Chapter 4, "Request processing," on page 79.

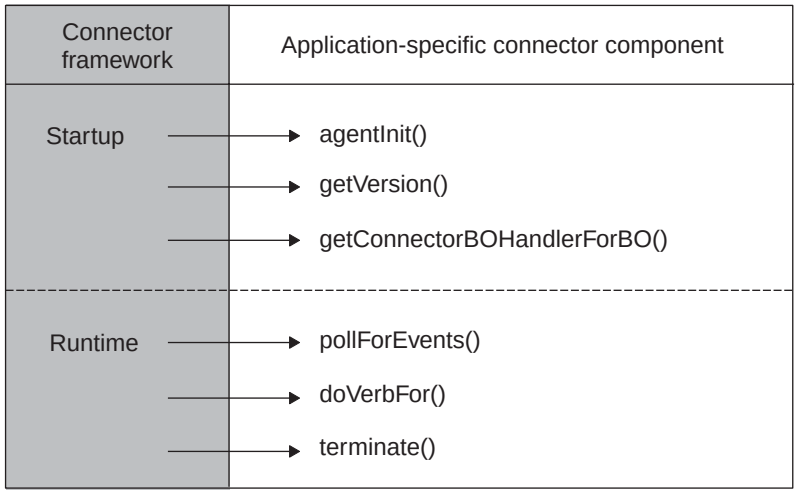

Figure 25. Summary of methods called by the connector framework

For more information on extending the connector base class, see "Extending the Java business-object-handler base class" on page 154.

Handling errors

The methods of the connector class library indicate error conditions in the following ways:

- Return codes—The connector class library includes a set of defined outcome-status values that your abstract methods can use to return information on the success or failure of a method. The return codes are defined as integer values and outcome-status constants. In your code, IBM recommends use of the predefined constants to prevent a problem if the IBM changes the values of the constants.

  For information on Java return codes, see "Java return codes" on page 203.
- Exceptions—The Java connector library provides classes to encapsulate exception objects and exception-detail objects, which contain exception information. For more information, see "Exceptions" on page 204.
- Return-status descriptor—during request processing, the connector framework sends status information back to the integration broker in a return-status descriptor. The business object handler can save a message and status code in this descriptor to provide the integration broker about the status of the verb processing. For more information, see "Return-status descriptor" on page 206.
- Error and message logging—The connector class library also provides the following features to assist in providing notification of errors and noteworthy conditions:
  - Logging allows you to send an informational or error message to a log destination.
  - Tracing allows you to include statements in your code that generate trace messages at different trace levels.

  For more information on how to implement logging and tracing, see Chapter 6, "Message logging," on page 139.

Using connector configuration property values

This section provides the following information about connector configuration properties:

- "What is a connector configuration property?"
- "Defining and setting connector configuration properties"
- "Retrieving connector configuration properties" on page 71

What is a connector configuration property?

A *connector configuration property* (sometimes called just a *connector property*) allows you to create named place holders (similar to variables) that the connector can use to access information it needs. Connectors have two categories of configuration properties:

- Standard configuration properties
- Connector-specific configuration properties

Standard connector configuration properties

Standard configuration properties provide information that is typically used by the connector framework. These properties are usually common to *all* connectors and usually represent well-defined behavior that is the WebSphere business integration system enforces.

Connector-specific configuration properties

Connector-specific configuration properties provide information needed by a particular connector at runtime. These configuration properties provide a way of changing static information or logic within the connector's application-specific component without having to recode and rebuild it. For example, configuration properties can be used to:

- Hold the value of constants, such as the name of the application server or database, the name of the event table, or the name of files the connector needs to read.
- Set behavior for the connector in a particular situation. For example, a configuration property can indicate that the connector should not fail a business object Retrieve operation for a hierarchical business object if a child object is missing. As another example, a configuration property can determine whether the application or the connector should create an ID for a new object on a Create operation.

You can create any number of connector-specific configuration properties for your connector. When you have identified needed connector-specific properties, you define them as part of the connector configuration process. Use Connector Configurator to specify connector configuration properties as part of the information stored in the local repository.

You can also add configuration properties later on as needed. In general, your connector code needs only to query for the values of the connector-specific properties such as `ApplicationUserID` and `ApplicationPassword`.

Defining and setting connector configuration properties

The Connector Configurator tool provides you with the ability to perform the following tasks on connector configuration properties:

- Assign a value to a standard configuration property.
- Define and assign a value to a connector-specific configuration property.

You invoke Connector Configurator from the System Manager tool.

WebSphere InterChange Server

If WebSphere InterChange Server is your integration broker, refer to the *Implementation Guide for WebSphere InterChange Server* for information about the Connector Configurator tool.

Other integration brokers

If a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) is your integration broker, refer the *Implementation Guide for WebSphere Message Brokers* for information about Connector Configurator. If WebSphere Application Server is your integration broker, refer to the *Implementation Guide for WebSphere Application Server* for information about Connector Configurator.

Retrieving connector configuration properties

Connector configuration properties are downloaded to the connector as part of the connector initialization (For more information, see "Starting up a connector" on page 63). Your connector application-specific component retrieves the values of any configuration properties that it needs for initialization based on the type of the connector property.

A connector can use a connector configuration property that has one of the following types:

- A *simple* connector configuration property contains only string values. It does *not* contain any other properties. A single-valued simple property contains only one string value.
- A *hierarchical* connector configuration property contains other properties and their values. A given connector property can contain multiple values.

Note: For the IBMWebSphere Business Integration Adapters product, single-valued simple connector configuration properties are the only kind of connector properties that a C++ connector supports. C++ connectors do *not* support hierarchical properties.

Note: In previous versions of the product, connector configuration properties were only single-valued and simple. That is, a connector property could contain only one string value. With this release, a Java connector can support hierarchical properties. As noted above, hierarchical properties can contain other properties and multiple values. Hierarchical properties are supported starting with version 2.2.0. For the IBM WebSphere InterChange Server product, this support starts with version 4.2.

Retrieving single-valued simple connector configuration properties

In previous versions of the product, connector configuration properties were *only* single-valued and simple. That is, a connector property could contain only *one* string value. To retrieve a single-valued simple connector configuration property, you can use the `getConfigProp()` method.

Note: For the IBMWebSphere Business Integration Adapters product, single-valued simple connector configuration properties were the only kind of connector properties supported in all releases *before* version 2.2.0. For the IBM WebSphere InterChange Server product, single-valued simple connector configuration properties were the only kind of connector properties supported in all releases *before* version 4.2. For backward compatibility, the mechanism described here to access single-valued simple connector properties is still supported by a Java connector. However, IBM recommends that new connector development use the mechanism described in "Retrieving hierarchical connector configuration properties" on page 73 to access connector configuration properties as hierarchical properties.

The Java connector library provides the two methods in Table 25 for retrieving the value of a simple connector configuration property.

Table 25. Methods for retrieving value of a simple connector configuration property

Connector library method	Description
`getConfigProp()`	Retrieves the value of a specified simple connector configuration property
`getAllConnectorAgentProperties()`	Retrieves the values of *all* connector configuration properties. However, if the method retrieves a multiple value connector property, it only retrieves the first of the connector-property values.

These methods are both defined in the `CWConnectorUtil` class and function as follows:

- The `getConfigProp()` method takes as input a string for the name of the configuration property and returns the value of this property as a Java `String`.
- The `getAllConnectorAgentProperties()` method does not require input arguments and returns the values of all connector configuration properties in a Java `Hashtable`.

The code fragment in Figure 26 uses the `getAllConnectorAgentProperties()` method to retrieve all connector configuration properties into a Java `Hashtable` object called `connectorProperties`. The code fragment then uses the `get()` method of the `Hashtable` class to retrieve the value of each connector configuration property.

```
connectorProperties =
    CWConnectorUtil.getAllConnectorAgentProperties();

// get Connector Configuration Properties to establish Connection
String connectString =
    (String)connectorProperties.get("ConnectString");
String userName =
    (String)connectorProperties.get("ApplicationUserName");
String userPassword =
    (String)connectorProperties.get("ApplicationPassword");

if(connectString == null || connectString.equals("")
        || userName==null || userPassword==null )
```

Figure 26. Retrieving all Connector Configuration Properties

Retrieving hierarchical connector configuration properties

A hierarchical connector configuration property can contain any of the following values:

- One or more child properties. Each child property can, in turn, contain its own child properties and string values.
- One or more string values.

A hierarchical connector property with more than one string value is called a *multi-valued* property. A property with only one string value is called a *single-valued* property.

The Java connector library represents a hierarchical connector configuration property with the `CWProperty class`. An object of this class is called a *connector-property object* and it can represent a simple or hierarchical, single- or multi-valued connector configuration property.

Table 26 lists the metadata for a hierarchical connector configuration property that a connector-property object provides.

Table 26. Metadata in a connector-property object

Connector-property information	Description	CWProperty method
Name	The name of the connector property	getName()
Cardinality	Indicates the number of values that the connector property contains: • single-valued • multi-valued	getCardinality()
Property type	Indicates whether the connector property contains any child properties: • simple: contains *no* child properties, only string values • hierarchical: contains one or more child properties	getPropType()
Encryption flag	Indicates whether the property value is to be encrypted.	getEncryptionFlag(), setEncryptionFlag()

As Table 26 indicates, retrieving metadata about the connector property is done with the methods indicated. However, retrieving the property *value* is a two-step process, as follows:

1. Retrieve the top-level connector-property object for one or all of the connector configuration properties.
2. Retrieve the desired property value from a connector-property object.

Retrieving the top-level connector-property object: To retrieve the top-level connector-property object for a connector property, you can use either of the methods in Table 27..

Table 27. Methods for retrieving top-level connector-property objects

Connector library method	Description
`getHierarchicalConfigProp()`	Retrieves the top-level connector-property object of a specified hierarchical connector configuration property

Table 27. Methods for retrieving top-level connector-property objects (continued)

Connector library method	Description
`getAllConfigProperties()`	Retrieves the top-level connector-properties objects for *all* connector configuration properties, regardless of whether the property is simple, hierarchical, or multi-valued.

The methods inTable 27 are both defined in the `CWConnectorUtil` class and function as follows:

- The `getHierarchicalConfigProp()` method takes the name of a connector configuration property as an argument. It returns a single `CWProperty` object that contains the top-level connector-property object for the specified connector property.
- The `getAllConfigProperties()` method returns an array of `CWProperty` objects, each containing a top-level connector-property object for one of the connector configuration properties.

Retrieving the connector-property value: Once you have retrieved the top-level connector-property object for a connector property, you can retrieve the values from this connector-property object. As discussed above, a hierarchical connector property can have one or more of the following kinds of values:

- One or more child properties
- One or more string values

Retrieving child properties: The `CWProperty` class provides the methods in Table 28 to retrieve child properties from a connector-property object.

Table 28. Methods for retrieving values child properties from a connector-property object

Description	CWProperty method
To obtain *all* child properties of the hierarchical connector property	getHierChildProps()
To obtain all child properties of the hierarchical connector property that has a specified prefix	getChildPropsWithPrefix()
To obtain a *single specified* child property from the hierarchical connector property	getHierChildProp()

You can use the `hasChildren()` method to determine whether the current connector-property object contains any child properties.

Retrieving string values: The `CWProperty` class provides the methods in Table 29 to retrieve string values from a connector-property object.

Table 29. Methods for retrieving values string values from a connector-property object

Description	CWProperty method
To obtain all string values of the hierarchical connector property	getStringValues()
To obtain all string values of a specified child property	getChildPropValue()

You can use the `hasValue()` method to determine whether the current connector-property object contains any string values.

Calling a data handler

The main task of a connector is to convert data between an application-specific form and a business object. Often, the connector must perform this conversion directly. For example, it can create the appropriate database statements to create or access the data as a row in a table of an application database. However, a connector might handle serialized data in a common Multipurpose Internet Mail Extensions (MIME) format.

Rather than have each connector perform the conversions between a particular MIME format and a business object, both the and WebSphere InterChange Server and WebSphere Business Integration Adapters products provide *data handlers* to perform these common conversions. A data handler is a special Java class instance that converts between serialized data in a particular MIME format and a business object. For example, the WebSphere Business Integration Data Handler for XML provides a data handler that converts between an XML document and business objects.

Note: This section provides a brief overview of data handlers. For a more complete description, see the *Data Handler Guide*.

The Java connector library provides several data-handler methods you can call a specific data handler from within the connector. To determine which data-handler method to use, you must perform the following tasks:

- "Determining direction of the data conversion"
- "Accessing the serialized data" on page 76
- "Identifying the data handler to instantiate" on page 77

Determining direction of the data conversion

A data handler can usually convert between serialized data and a business object in both directions; that is, it can perform both of the following conversions:

- *String-to-business-object conversion* converts serialized data to a business object.

  Within a connector, this conversion is useful during event processing, when the connector receives serialized data from the application and must create the appropriate business-object representation of this data, which it then sends to the integration broker. The connector can send the business object to the appropriate data handler and receive from it the corresponding serialized data (as long as a data handler exists to convert to the desired format of serialized data).
- *Business-object-to-string conversion* converts a business object to serialized data.

  This conversion is useful during request processing when the connector receives a business object from the integration broker and must create the appropriate serialized data, which it then sends to the application. The connector can send in the business object to the appropriate data handler and receive the corresponding business object (as long as a data handler exists to convert the desired format for the serialized data).

The Java connector library provides the data-handler methods in Table 30 so that a connector can call a data handler to convert between serialized data in a particular MIME format and a business object. These methods are defined in the `CWConnectorUtil` class.

Table 30. Data-handler methods in Java Connector Library

Conversion	Conversion process	Method
Business-object-to-string	Call a data handler to convert the specified business object (*theBusObj* argument) to serialized data, returning this data in one of the supported access forms. For more information, see "Accessing the serialized data.".	"boToByteArray()" on page 349 "boToStream()" on page 351 "boToString()" on page 353
String-to-business-object	Call a data handler to convert the specified serialized data (the *serializedData* argument) to a business object.	"byteArrayToBo()" on page 355 "readerToBO()" on page 372 "streamToBO()" on page 374 "stringToBo()" on page 376

If the data handler cannot perform the requested conversion, the data-handler method throws the `ParseException` exception.

Accessing the serialized data

To access the serialized data sent to or received from a data-handler method, you must provide the following information:

- In what format your code will access the serialized data
- In what locale the serialized data exists

Choosing a data format

The purpose of a data handler is to convert between serialized data and a business object. Therefore, the code of the Java connector must be able to access to this serialized data. It might have access to this data in any of the forms listed in Table 31 The Java connector library provides data-handler methods that support each of these forms of serialized data.

Table 31. Ways to access serialized data to and from data handlers

Access to serialized data	Java construct	Method
A string	`String` object	boToString() stringToBo()
An input stream	An object of `java.io.InputStream` class or one of its subclasses	boToStream() streamToBO()
A reader for character streams	An object of `java.io.Reader` class or one of its subclasses	readerToBO()
A byte array	`byte[]`	boToByteArray() byteArrayToBo()

To access the serialized data sent to or received from a data handler, choose the data-handler method from Table 31 that handles the appropriate access format.

Identifying the data locale and encoding

As shown in Table 30., the data-handler methods call a data handler to either read serialized data (string-to-business-object conversion) or create serialized data (business-object-to-string conversion). During this process, the data handler might

need to know about the character encoding or locale of the serialized data it is processing. To allow you to specify a different locale or character encoding for the data handler to use, the data-handler methods accept a Java `Locale` object and a `String` encoding argument to specify this information:

- If the locale is the *same* as the connector-framework locale, you can specify a `null` for the *locale* argument in the call to the data-handler method. If the locale is different, specify a `java.util.Locale` object that contains the appropriate locale information.
- If the character encoding is the *same* as that the connector framework is using, you can specify a `null` for the *encoding* argument in the call to the data-handler method. If the character encoding is different, specify the appropriate character encoding as a Java `String`.

For information on how to obtain the connector-framework locale or character encoding, see "Design considerations for an internationalized connector" on page 56.

Identifying the data handler to instantiate

To identify the data handler that needs instantiation, the data-handler methods must provide the instantiation process with the information it needs to locate the data handler's class. This data-handler class is the name of the Java class that implements the data handler.

Note: The data-handler methods must instantiate a data handler *before* they can request the specified conversion. This instantiation process is implemented by the `createHandler()` method of the `DataHandler` base class. For more information on the `DataHandler` class and the data-handler configuration information, see the*Data Handler Guide*.

The data-handler method can specify the name of the data-handler class by providing the MIME type of the serialized data in its *mimeType* argument and, optionally its *BOPrefix* argument. It uses this MIME type to obtain the data handler's class from its child meta-object in the top-level meta object as follows:

- The data-handler method checks the top-level meta-object for the data handler that corresponds to this specified MIME type. It obtains the name of this top-level meta-object from the `DataHandlerMetaObjectName` connector configuration property. If this property is not set, the data-handler method throws the `PropertyNotSetException` exception.
- The top-level meta-object contains attributes whose names indicate supported MIME types. The attribute types identify the child meta-object that corresponds to the specified MIME type. This child meta-object contains configuration information for the data handler, including the data handler's class name.

In this case, the data-handler method instantiates a data handler of Java class listed in the child meta-object. The instantiation process uses the child data-handler meta-object associated with that MIME type to derive the class name and other configuration information for the data handler instance.

Note: Each system on which data handlers are installed has a meta-object to describe the available data handlers. A meta-object is a special business object that contains configuration information. For data handlers, the top-level meta-object contains the available data handlers and the associated MIME type that each data handler supports.

For more information about the meta-objects and about how the instantiation process derives a class name from the specified MIME type, see the *Data Handler Guide*.

If the data handler cannot be instantiated, the data-handler method throws the `DataHandlerCreateException`.

Handling loss of connection to an application

A good design practice is to code the connector application-specific code so that it shuts down whenever the connection to the application is lost. To respond to a lost connection, the connector's application-specific component should take the following steps:

- Log a fatal error message so that email notification is triggered if the `LogAtInterchangeEnd` connector configuration property is set to True.
- Return the `APPRESPONSETIMEOUT` outcome status inform the connector controller that the application is not responding. When this occurs, the process in which the connector runs is stopped. A system administrator must fix the problem with the application and restart the connector to continue processing events and business object requests.

The following user-implemented abstract methods should check for a loss of connection to the application:

- For event notification, the `pollForEvents()` method should verify the connection before it accesses the event store. For more information, see "Verifying the connection before accessing the event store" on page 182.
- For request processing, the `doVerbFor()` method should verify the connection before it begins verb processing. For more information, see "Verifying the connection before processing the verb" on page 158.

Chapter 4. Request processing

This chapter presents information on how to provide request processing in a connector. *Request processing* implements a mechanism to receive requests, in the form of request business objects, from an integration broker and to initiate the appropriate changes in the application business entities. The mechanism for implementing request processing is a *business object handler*, which contains methods that interact with an application to transform request business objects into requests for application operations. This chapter contains the following sections:

Note: For an introduction to request processing, see "Request processing" on page 24..

Designing business object handlers

The business object handler implements request processing for the connector. Therefore, the defining and coding of business object handlers is one of the primary tasks in connector development. A business object handler is an instance of a subclass of the `CWConnectorBOHandler` class. Each business object definition refers to a business object handler, which contains a set of methods to perform the tasks for the verbs that the business object definition supports. You need to code one or more business object handlers to process the business objects that the connector supports.

The way to implement a business object handler depends on the application programming interface (API) that you are using and how this interface exposes application entities. To determine how many business object handlers your connector requires, you need to take a look at the application that the connector will interact with:

- If the application is form-based, table-based, or object-based and has a standard access method across entities, you might be able to design business objects that store information about application entities. The business object handler can process the application entities in a *metadata-driven business object handler*.

  You can derive one generic business-object-handler class to implement a metadata-driven business object handler, which handles processing of *all* business objects. For more information, see "Implementing metadata-driven business object handlers" on page 80.

- If the application has different access methods for different kinds of entities, some or all of the application entities might require individual business object handlers.

  You can:

  – Derive a generic business-object-handler class to implement a metadata-driven business object handler for some business objects, and separate business-object-handler classes to implement business object handlers for other business objects.
  – Derive multiple business-object-handler classes, one for each business object definition that the connector supports.

  For more information, see "Implementing multiple business object handlers" on page 81.

Another consideration in the design of a business object handler is whether you need to have separate processing for certain verbs of the business object. If some verb (or verbs) require special processing, you can create a custom business object handler for the verb. For more information, see "Creating a custom business object handler" on page 174."Creating a custom business object handler" on page 184.

Implementing metadata-driven business object handlers

If the application API is suitable for a metadata-driven connector, and if you design business object definitions to include metadata, you can implement a metadata-driven business object handler. This business object handler uses the metadata to process all requests. A business object handler can be completely metadata-driven if the application is consistent in its design, and the metadata follows a consistent syntax for each supported business object.

Note: For an introduction to metadata and metadata-driven design, see "Assessing support for metadata-driven design" on page 47..

This section provides the following information about metadata-driven design for a business object handler:

- "Metadata in business objects"
- "Benefits of metadata design" on page 81

Metadata in business objects

Business object definitions have specific locations for different types of application-specific data. For example, business object attributes have a set of properties, such as Key, Foreign Key, Required, Type, and so on, that provide the business object handler with information that it can use to drive business object processing. In addition, the `AppSpecificInfo` property can provide the business object handler with application-specific information, which can specify how to access data in the application and how to process application entities.

The `AppSpecificInfo` property is available for the business object definition, attributes, and verbs. Table 32 shows some typical schemes for encoding application-specific information in business objects.

Table 32. Example schemes for storage of application information in business objects

Scope of application-specific information	Table-based application	Form-based application
The whole business object	Table name	Form name
An individual attribute	Column name	Field name

Table 32. Example schemes for storage of application information in business objects (continued)

Scope of application-specific information	Table-based application	Form-based application
The business object verb	SQL statement or other verb-processing instructions	Action to be performed

Using application-specific information, a metadata-driven business object handler might simply:

- Examine the verb of an incoming business object to identify the operation to perform.
- Examine the contents of the business object metadata to identify the name of the associated application entity (such as an application table or form).
- Examine the contents of the attribute metadata to identify fields, columns, or other information about the attributes.

If a business object definition contains the table name and column names, you do not have to explicitly code those names in the business object handler.

Benefits of metadata design

Encoding application information in a business object accomplishes two things:

- One business object handler class can perform *all* operations for *all* business objects supported by the connector. You do not have to code a separate business object handler for each supported business object.
- Changes to a business object definition do not require recoding the connector as long as the changes conform to existing metadata syntax. This benefit means that you can add attributes to a business object definition, remove attributes, or reorder attributes without recompiling or recoding the connector.

If information about application entities is encoded consistently in the business object definition, all request business objects can be handled by a single business-object-handler class in the connector. Also, you need to implement only a single `getConnectorBOHandlerForBO()` method to return the single business object handler and a single `doVerbFor()` method to implement this business object handler. This approach is recommended for connector development because it provides flexibility and automatic support for new business object attributes.

Implementing multiple business object handlers

For each business object definition that does *not* encapsulate all the metadata and business logic for an application entity, you need a separate business-object-handler class. You can derive separate handler classes directly from the business-object-handler base class, or you can derive a single utility class and derive subclasses as needed. You must then implement the `getConnectorBOHandlerForBO()` method to return business object handler that corresponds to particular business object definitions.

Each business object handler must contain a `doVerbFor()` method. If you implement multiple business object handlers, you must implement a `doVerbFor()` method for each business-object-handler class. In each `doVerbFor()` method, include code to handle any parts of the application entity or operations on the application entity that the business object definition does not describe.

This approach results in higher maintenance requirements and longer development time than designing a single business object handler for a metadata-driven

connector. For this reason, this approach should be avoided if possible. However, if the application has different access methods for different kinds of entities, coding multiple, entity-specific business object handlers might be unavoidable.

Extending the business-object-handler base class

The Java connector library provides the business-object-handler base class, `CWConnectorBOHandler`. This base class includes methods for handling request processing, including the `doVerbFor()` method. To create a business object handler, you must extend this business-object-handler base class and implement its abstract method `doVerbFor()`. For information specific to the Java connector library, see "Extending the Java business-object-handler base class" on page 154."Extending the Java business-object-handler base class" on page 164.

Handling the request

Once you have derived your business-object-handler class, you must implement the business-object-handler method, `doVerbFor()`. It is the `doVerbFor()` method that provides request processing for the business objects that the connector supports. At startup, the connector framework calls `getConnectorBOHandlerForBO()` to obtain the business object handler implemented for each of the business object definitions that the connector supports.

Important: All connectors *must* implement a business-object-handler method, `doVerbFor()`, that implements the Retrieve verb. This method and verb *must* be implemented even if your connector will *not* perform request processing.

This section provides the following information on how to implement the `doVerbFor()` method:

- "Basic logic for doVerbFor()"
- "General recommendations on verb implementations" on page 84

Basic logic for doVerbFor()

For a Java connector, the `CWConnectorBOHandler` class defines the `doVerbFor()` method, which is an abstract method defined. The `doVerbFor()` method typically follows a basic logic for request processing.

Figure 27 shows a flow chart of the method's basic logic.

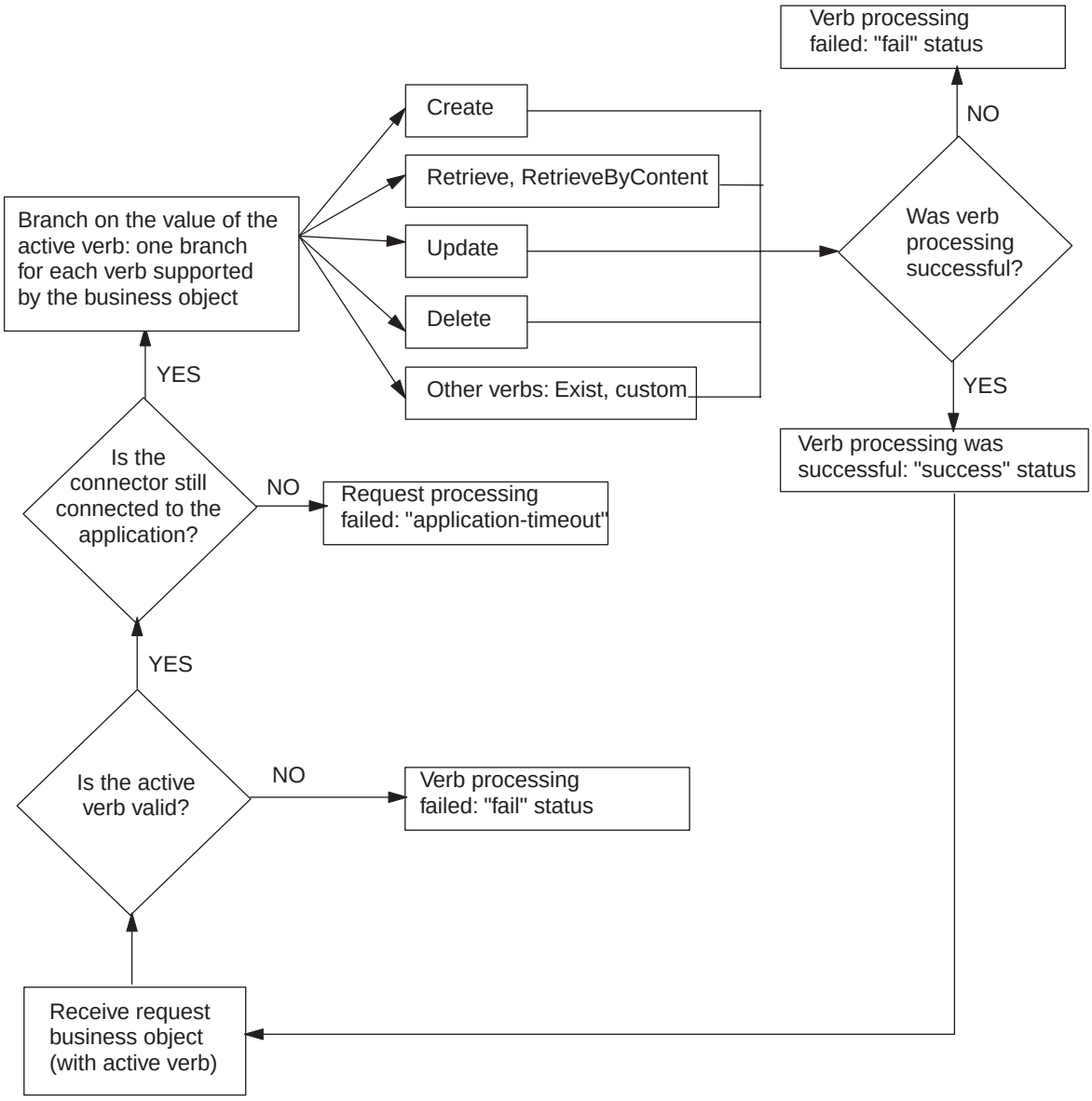

Figure 27. Flow chart for basic logic of doVerbFor()

For an implementation of this basic `doVerbFor()` logic, see "Implementing the doVerbFor() method" on page 155."Implementing the doVerbFor() method" on page 165.

When the connector framework receives a request, it calls the `doVerbFor()` method for the business-object-handler class associated with the business object definition of the request business object. To this `doVerbFor()` method, the connector framework passes the request business object. Table 33 summarizes the tasks that the `doVerbFor()` method performs once it has received a request business object from the connector framework.

Table 33. Tasks of the doVerbFor() method

Task of business object handler		For more information
1.	Determine the verb processing to perform, based on the active verb in the request business object.	"Performing the verb action" on page 85

Table 33. Tasks of the doVerbFor() method (continued)

Task of business object handler	For more information
2. Obtain information from the request business object to build and send requests for operations to the application.	"Processing business objects" on page 106

General recommendations on verb implementations

This section provides the following general recommendations for implementing your `doVerbFor()` method:

- "Verb stability"
- "Transaction support"
- "ObjectEventId attribute"

Verb stability

Verbs in a business object should remain stable throughout the request and response cycle. When a connector receives a request, the hierarchical business object that is returned to InterChange Server should have the same verbs as the original request business object, with the exception of verbs in child business objects that were *not* set in the original request.

Verbs in child business objects might or might not be set in request business objects:

- When a verb is set in a child business object, the connector should perform the operation that the child verb indicates, regardless of the verb on the top-level business object.
- If a verb in a child business object request is *not* set, the connector can either leave the child verb as `NULL`, set the child verb to the verb in the top-level business object, or set the value of the verb to the operation that the connector needs to perform.

Transaction support

An entire business object request must be wrapped in a single transaction. In other words, all Create, Update, and Delete transactions for a top-level business object and all of its children must be wrapped in a single transaction. If any failure is detected during the life of the transaction, the whole transaction should be rolled back.

For example, if a Create operation on a top-level business object succeeds, but the transaction for one of the child business objects fails, the connector application-specific component should roll back the entire Create transaction to the previous state. In this case, the connector's application-specific component should return failure from the verb method.

ObjectEventId attribute

The `ObjectEventId` attribute is used in the IBM WebSphere business integration system to identify an event-trigger flow in the system. In addition, it is used to keep track of child business objects across requests and responses, as the position of child business objects in a hierarchical business object request might be different from the position of the child business objects in the response business object.

Connectors are *not* required to populate `ObjectEventId` attributes for either a parent business object or its children. If business objects do not have values for `ObjectEventId` attributes, the IBM WebSphere business integration system

generates values for them. When connectors generate ObjectEventId values, this is done by the source connector as part of the event-notification mechanism.

When processing request business objects, connectors should preserve ObjectEventId values in *all* levels of a hierarchical business object between the request business object and the response business object. If a connector method changes the values of child business object ObjectEventIds, the IBM WebSphere business integration system may not be able to correctly track the child business objects.

For information on generating ObjectEventIds in the event notification mechanism, see "Event identifier" on page 117..

Performing the verb action

The standard verbs that IBM WebSphere business integration system expect connectors to handle are Create, Retrieve, Update, and Delete. IBM recommends that you implement these verbs according to standard behaviors documented in the sections listed in the For More Information column of Table 34.. These sections provide information about the standard behavior, implementation notes, and the appropriate outcome-status values.

Table 34 lists the standard verbs that IBM WebSphere business integration system defines. Your doVerbFor() method should implement those verbs appropriate for its application.

Table 34. Verbs implemented by the doVerbFor() method

Verb	Description	For more information
Create	Make a new entity in the application.	"Handling the Create verb" on page 86
Retrieve	Using key values, return a complete business object.	"Handling the Retrieve verb" on page 89
RetrieveByContent	Using non-key values, return a complete business object.	"Handling the RetrieveByContent verb" on page 95
Update	Change the value in one or more fields in the application.	"Handling the Update verb" on page 97
Delete	Remove the entity from the application. This operation must be a true physical delete.	"Handling the Delete verb" on page 104
Exists	Check whether the entity exists in the application.	"Handling the Exists verb" on page 105
Custom verbs	Perform some application-specific operation.	None

Note: Although the sections listed in the "For more information" column of Table 34 present suggested behavior for verb methods, your connector might need to implement some aspects of verb processing differently to support a particular application. Once the connector framework passes a request business object to your connector's doVerbFor() method, the doVerbFor() method can implement verb processing in any way that is necessary. Your verb processing code is not limited to the suggestions presented in this chapter.

InterChange Server

When InterChange Server is the integration broker and you design your own collaborations, you can implement any custom verbs that you need. Your

collaborations and connectors are not limited to the standard list of verbs.

End of InterChange Server

This basic verb-processing logic consists of the following steps:

1. Get the verb from the request business object.

   The `doVerbFor()` method must first retrieve the active verb from the business object with the `getVerb()` method. For a Java connector, `getVerb()` is defined in the `CWConnectorBusObj` class.

2. Perform the verb operation.

   In the business object handler, you can design the `doVerbFor()` method in either of the following ways:

   - Implement verb processing for each supported verb directly within the `doVerbFor()` method. You can modularize the verb processing so that each verb operation is implemented in a separate verb method called from `doVerbFor()`. The method should also take appropriate action if the verb is not a supported verb by returning a message in the return-status descriptor and a "fail" status.
   - Handle all verb processing in the same method using a metadata-driven `doVerbFor()` method.

Handling the Create verb

When the business object handler obtains a Create verb from the request business object, it must ensure that a new application entity, whose type is indicated by the business object definition, is created, as follows:

- For a flat business object, the Create verb indicates that the specified entity must be created.
- For a hierarchical business object, the Create verb indicates that one or more application entities (to match the entire business object) must be created.

The business object handler must set all the values in the new application entities to the attribute values in the request business object. To ensure that all required attributes in the request business object have values assigned, you can call the `initAndValidateAttributes()` method, which assigns the attribute's default value to each required attribute that does not have its value set (when the `UseDefaults` connector configuration property is set to `true`). The `initAndValidateAttributes()` method is defined in the `CWConnectorUtil` class. Call `initAndValidateAttributes()` *before* performing the Create operation in the application.

Note: For a table-based application, the entire application entity must be created in the application database, usually as a new row to the database table associated with the business object definition of the request business object.

This section provides the following information to help process a Create verb:

- "Standard processing for a Create verb" on page 87
- "Implementation of a Create verb operation" on page 87
- "Outcome status for Create verb processing" on page 88

Note: You can modularize your business object handler so that each supported verb is handled in a separate Java method. If you follow this structure, a Create method handles processing for the Create verb.

Standard processing for a Create verb

The following steps outline the standard processing for a Create verb:

1. Create the application entity corresponding to the top-level business object.
2. Handle the primary key or keys for the application entity:
   - If the application generates its own primary key (or keys), get these key values for insertion in the top-level business object.
   - If the application does *not* generate its own primary key (or keys), insert the key values from the request business object into the appropriate key column (or columns) of the application entity.
3. Set foreign key attributes in any first-level child business objects to the value of the top-level primary key.
4. Recursively create the application entities corresponding to the first-level child business objects, and continue recursively creating all child business objects at all subsequent levels in the business object hierarchy.

In Figure 28,, a verb method sets the foreign key attributes (FK) in child business objects A, B, and C to the value of the top-level primary key (PK1). The method then recursively sets the foreign key attributes in child business objects D and E to the value of the primary key (PK3) in their parent business object, object B.

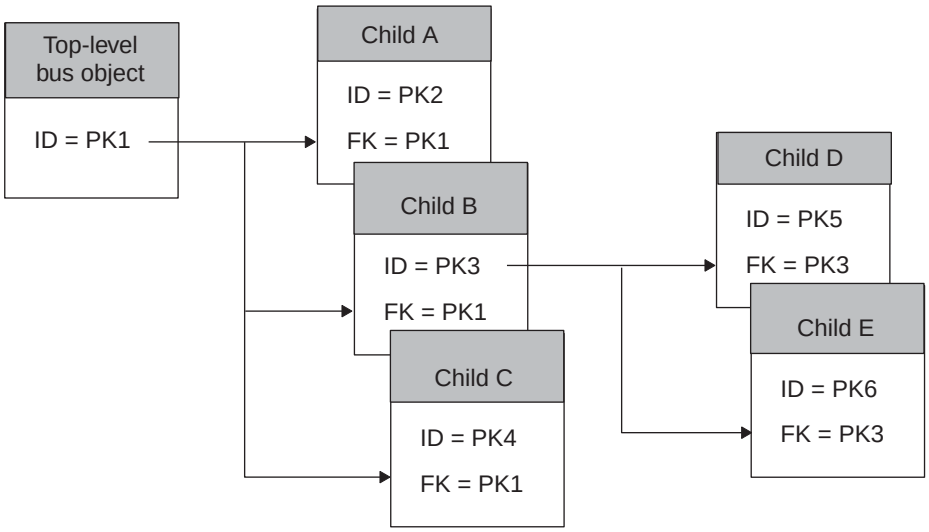

Figure 28. Creating parent/child relationships

Implementation of a Create verb operation

A typical implementation of a Create operation first traverses the top-level business object and processes the business object's simple attributes. It gets the values of the attributes from the business object and generates the application-specific action (such as an API call or SQL statement) that inserts an entity in the application to represent the top-level business object. Once this top-level entity is created, the verb operation takes the following steps:

1. Retrieve any primary keys for the entity from the application.
2. Use the keys to set the foreign key attributes in the first-level child business objects to the value of the parent primary keys.
3. Set the verb in each child business object to Create and recursively create application entities to represent the child business objects.

A recommended approach for creating child business objects is to design a submethod to recursively create child entities. The submethod might traverse the business object, looking for attributes of type `OBJECT`. If the submethod finds attributes that are objects, it calls the main Create method to create the child entities.

The way that the main method provides primary key values to the submethod can vary. For example, the main Create method might pass the parent business object to the submethod, and the submethod can then retrieve the primary key from the parent business object to set the foreign key in the child business object. Alternatively, the main method might traverse the parent object, find first-level children, set the foreign key attributes in the child business objects, and then call the submethod on each child.

In either case, the main Create method and its submethod interact to set the interdependencies between the parent business object and its first-level children. Once the foreign keys are set, the operation can:

- Insert new rows into the application.
- Set foreign keys for the next level of child business objects.
- Create the child entities.
- Descend the business object hierarchy, recursively creating child entities until there are no more child business objects to process.

Note: For a table-based application, the order of the steps for setting the relationships between a top-level object and its children may vary, depending on the database schema for the application and on the design of the application-specific business objects. For example, if foreign keys for a hierarchical business object are located in the top-level business object, the verb operation might need to process all child business objects *before* processing the top-level business object. Only when the child entities are inserted into the application database and the primary keys for these entities are returned can the top-level business object be processed and inserted into the application database. Therefore, be sure to consider the structure of data in the application database when you implement connector verb methods.

Outcome status for Create verb processing

The Create operation should return one of the outcome-status values shown in Table 35..

Table 35. Possible outcome status for Java Create verb processing

Create condition	Java outcome status
If the Create operation is successful and the application generates new key values, the connector: • fills the business object with the new key values; this business object is returned to the connector framework through the request business object parameter. • returns the "Value Changed" outcome status to indicate that the connector has changed the business object	`VALCHANGE`
If the Create operation is successful and the application does *not* generate new key values, the connector can simply return "Success".	`SUCCEED`
If the application entity already exists, the connector can *either* of the following actions: • Fail the Create operation.	`FAIL`

Table 35. Possible outcome status for Java Create verb processing (continued)

Create condition	Java outcome status
• Return an outcome status that indicates the application entity already exists.	VALDUPES
If the Create operation fails, the verb operation: • fills a return-status descriptor with information on the failure • returns the "Fail" outcome status	FAIL

Note: When the connector framework receives the VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 169."Sending the verb-processing response" on page 179.

Handling the Retrieve verb

When the business object handler obtains a Retrieve verb from the request business object, it must ensure that an existing application entity, whose type is indicated by the business object definition, is retrieved, as follows:

- For a flat business object, the Retrieve verb indicates that the specified entity is retrieved by its key values. The verb operation returns a business object that contains the current values for the application entity.
- For a hierarchical business object, the Retrieve verb indicates that one or more application entities (to match the entire business object) are retrieved by the key values of the top-level business object. The verb operation returns a business object in which all simple attributes of each business object in the hierarchy match the values of the corresponding entity attributes, and the number of individual business objects in each child business object array matches the number of child entities in the application.

Note: For a table-based application, the entire application entity must be retrieved from the application database.

For the Retrieve verb, the business object handler obtains the key value (or values) from the request business object. These key values uniquely identify an application entity. The business object handler then uses these key values to retrieve all the data associated with an application entity. The connector retrieves the entire hierarchical image of the entity, including all child objects. This type of retrieve operation might be referred to as an *after-image retrieve*.

Important: All connectors must implement a `doVerbFor()` method with verb processing for the Retrieve verb. This requirement holds even if your connector will *not* perform request processing.

An alternative way of retrieving data is to query using a subset of non-key attribute values, none of which uniquely define a particular application record. This type of retrieve processing is performed by the RetrieveByContent verb method. For information on retrieving by non-key values, see "Handling the RetrieveByContent verb" on page 95.

This section provides the following information to help process a Retrieve verb:

- "Standard processing for a Retrieve verb" on page 90
- "Implementation of a Retrieve verb operation" on page 90

- "Example: Retrieve operation"
- "Retrieving child objects" on page 92
- "Outcome status for Retrieve verb processing" on page 94

Note: You can modularize your business object handler so that each supported verb is handled in a separate Java method. If you follow this structure, a Retrieve method handles processing for the Retrieve verb.

Standard processing for a Retrieve verb

The following steps outline the standard processing for a Retrieve verb:

1. Create a new business object of the same type as the request business object. This new business object is the *response business object*, which will hold the retrieved copy of the request business object.
2. Set the primary keys in the new top-level business object to the values of the top-level keys in the request business object.
3. Retrieve the application data for the top-level business object and fill the response top-level business object's simple attributes.
4. Retrieve *all* the application data associated with the top-level entity, and create and fill child business objects as needed.

Note: By default, the Retrieve method returns failure if it cannot retrieve application data for *all* the child objects in a hierarchical business object. This behavior can be made configurable; for information, see "Configuring a Retrieve to ignore missing child objects" on page 94.

Implementation of a Retrieve verb operation

A typical Retrieve operation can use one of the following methods:

- Create a new response business object from the business object definition for that object and sets the top-level primary keys in this new business object. Using the top-level primary keys, the verb operation can retrieve all data associated with the top-level entity.
- Start by pruning all child business objects from the top-level business object. Using the top-level keys in the pruned object, the verb operation can retrieve the top-level data and all associated data.

The goal of each of these approaches is the same: Start with the top-level business object and rebuild the complete business object hierarchy. This type of implementation ensures that *all* children in the request business object that are no longer in the database are removed and are not passed back in the response business object. This implementation also ensures that the hierarchical response business object exactly matches the database state of the application entity. At each level, the Retrieve operation rebuilds the request business object so that it accurately reflects the current database representation of the entity.

Example: Retrieve operation

In a Retrieve operation, an integration broker requests the complete set of data that is associated with an application entity. The request business object might contain any of the following:

- A top-level business object but no child objects, even though the business object definition includes children
- A business object that contains the top-level business object and some of its defined children

- A complete hierarchical business object containing all child business objects

Figure 29 shows a request business object for a Contact entity. The `Contact` business object includes a multiple cardinality array for the `ContactProfile` attribute. In this request business object, the `ContactProfile` business object array includes two child business objects.

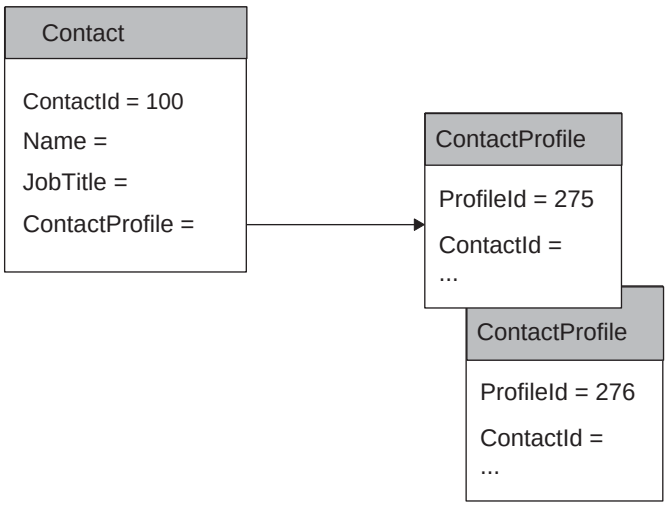

Figure 29. Example business object content for a Retrieve request

Application tables associated with the `Contact` and `ContactProfile` business objects might look like the tables in Figure 30.. This illustration also shows the foreign-key relationship between the tables. As you can see, the `contact_profile` table has a row for the `ContactId` of 100 that is *not* reflected in the `Contact` request business object in Figure 28..

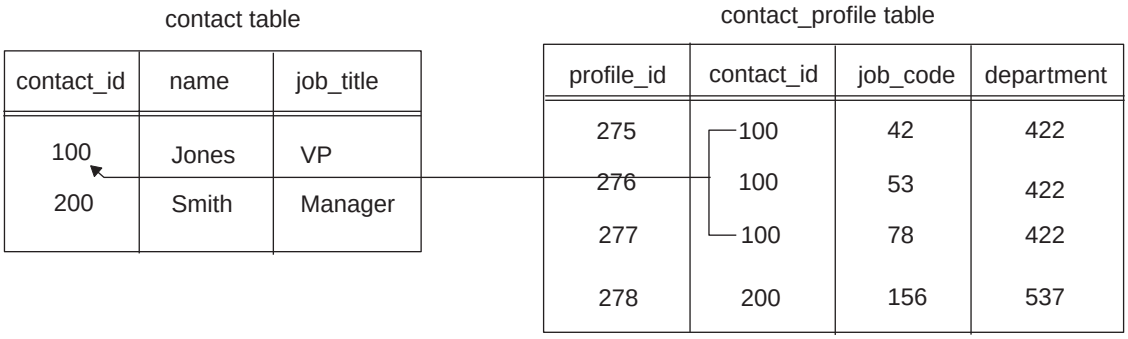

Figure 30. Foreign-key relationships between tables

The Retrieve operation uses the primary key in the `Contact` business object (100) to retrieve the data for the simple attributes in the response business object: values for the `Name` and `JobTitle` attributes. To be sure that it retrieves the correct number of child business objects, the verb operation must either create a new business object or prune child objects from the existing request business object. For the tables in Figure 30,, the Retrieve operation would need to create a new `ContactProfile` business object for the `contact_profile` row with a `profile_id` value of 277. In this way, the Retrieve operation properly creates and populates *all* arrays based on the current state of the application entities.

Retrieving child objects

To retrieve entities associated with the top-level entity, the Retrieve operation might be able to use the application API:

- Ideally, the API will navigate the relationships between application entities and return all related data. The verb operation can then encapsulate the related data as child business objects.
- If the API does *not* provide information on associated entities, you might need to access the application (for example, with generated SQL statements) to retrieve related data. The SQL statements might use foreign keys to navigate through application tables.

If the attribute application-specific information in the business object definition contains information on foreign keys, the verb operation can use this information to generate command to access the application (such as SQL statements). For example, application-specific information for the foreign key attribute of the `ContactProfile` child business object might specify:

- The parent table: `contact`
- The child table's column for the foreign key: `contact_id`
- The attribute in the parent business object that contains the primary key value that serves as a foreign key in the child business object: `ContactId`

Figure 31 shows example application-specific information for the primary key attribute of the Contact business object and the primary and foreign key attributes of the `ContactProfile` child business object.

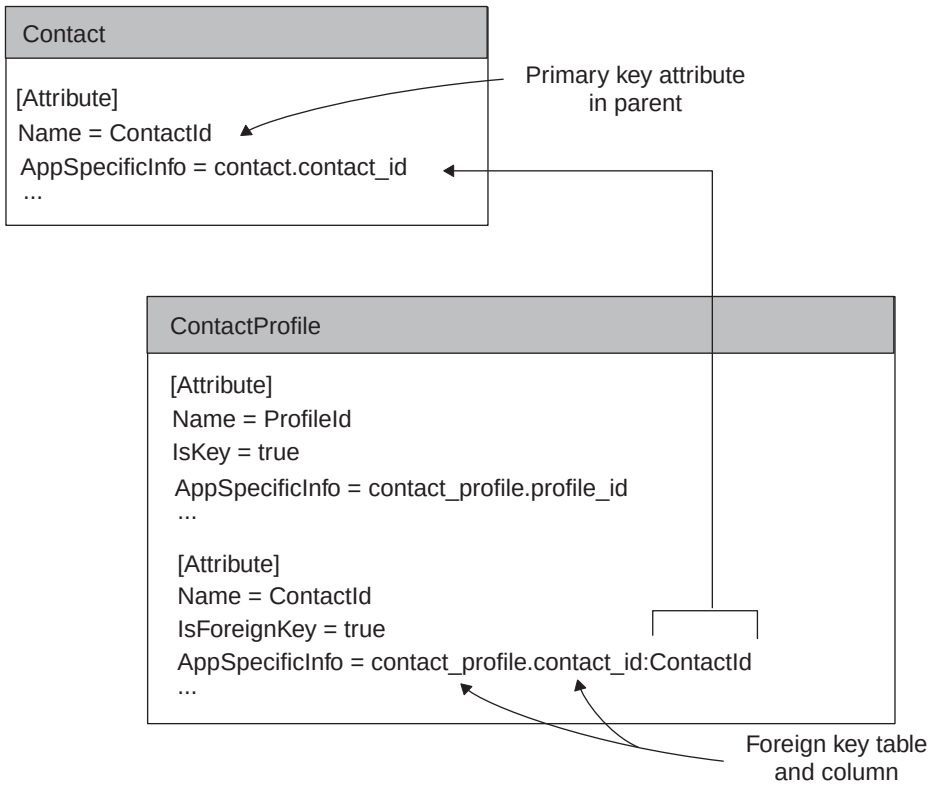

Figure 31. Foreign-key relationships in business objects

Using the application-specific information, the verb operation can find the name of the child table (`contact_profile`) and the column for the foreign key (`contact_id`) in the child table. The verb operation can also find the value of the foreign key for the child business object by obtaining the value of the primary key attribute (`ContactId`) in the parent business object (100). With this information, the verb operation can generate a SQL SELECT statement that retrieves all the records in the child table associated with the parent key. The SELECT statement to retrieve the data associated with the missing `contact_profile` row might be:

```
SELECT profile_id, job_code, department
FROM contact_profile
WHERE contact_id = 100
```

The SELECT statement returns three rows from the example `contact_profile` table, as shown in Figure 32..

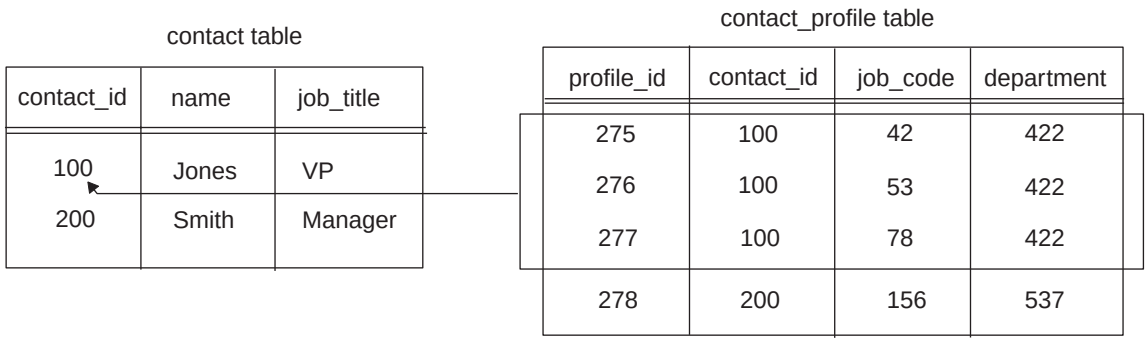

Figure 32. Results of SELECT statement for example Retrieve operation

If a Retrieve operation returns multiple rows, each row becomes a child business object. The verb operation might process retrieved rows as follows:

1. For each row, create a new child business object of the correct type.
2. Set attributes in the new child business object based on the values that a SELECT statement returns for the associated row.
3. Recursively retrieve all children of the child business object, creating the business object and setting the attributes for each one.
4. Insert the array of child business objects into the multiple-cardinality attribute in the parent business object.

The response business object for the Retrieve operation on the two example tables might look like Figure 33.. The verb operation has retrieved the current database entity and has added a child to the hierarchical business object.

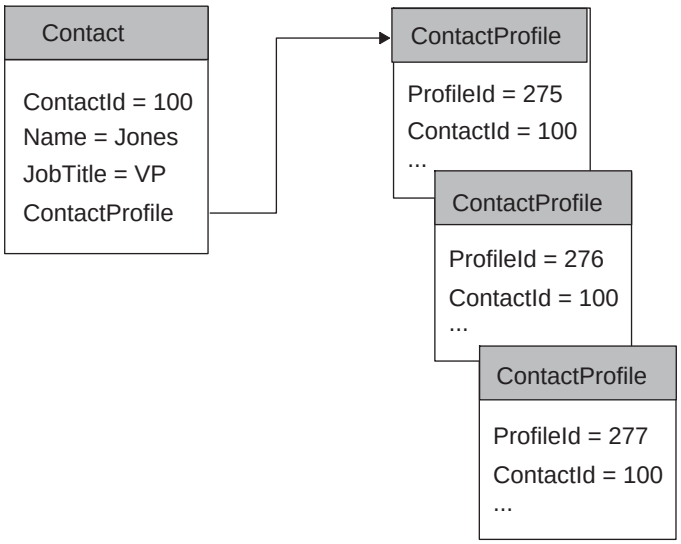

Figure 33. Business object response to example Retrieve request

Configuring a Retrieve to ignore missing child objects

By default, the Retrieve operation should return failure if it cannot retrieve application data for the complete set of child business objects in a hierarchical business object. However, you can implement the verb operation so that the behavior of the connector is configurable when one or more of the children in a business object are not found in the application.

To do this, define a connector-specific configuration property named `IgnoreMissingChildObject`, whose values are True and False. The Retrieve operation obtains the value of this property to determine how to handle missing child business objects. When the property is True, the Retrieve operation should simply move on to the next child in the array if it fails to find a child business object. In this case, the verb operation should return VALCHANGE if it is successful in retrieving the top-level object, regardless of whether it is successful in retrieving its children.

Outcome status for Retrieve verb processing

The Retrieve operation should return one of the outcome-status values shown in Table 36..

Table 36. Possible outcome status for Java Retrieve verb processing

Retrieve condition	Java outcome status
When the Retrieve operation is successful, it: • fills the entire business object hierarchy, including all child business objects; this business object is returned to the connector framework through the request business object parameter. • returns the "Value Changed" outcome status to indicate that the connector has changed the business object	`VALCHANGE`

Table 36. Possible outcome status for Java Retrieve verb processing (continued)

Retrieve condition	Java outcome status
If the `IgnoreMissingChildObject` connector property is True, the Retrieve operation returns the "Value Changed" outcome status for the business object if it is successful in retrieving the top-level object, regardless of whether it is successful in retrieving its children.	`VALCHANGE`
If the entity that the business object represents does *not* exist in the application, the connector returns a special outcome status instead of "Fail".	`BO_DOES_NOT_EXIST`
If the request business object does *not* provide a key for the top-level business object, the Retrieve operation can take *either* of the following actions: • Fill a return-status descriptor with information about the cause of Request failure and return a "Fail" outcome status. • Call the RetrieveByContent method to retrieve using the content of the top-level business object.	`FAIL`

Note: When the connector framework receives the VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 169."Sending the verb-processing response" on page 179.

Handling the RetrieveByContent verb

An integration broker might need to retrieve a business object for which it has a set of attribute values without having the key attribute (or attributes) that uniquely identifies an application entity. Such a retrieve is called "retrieve by non-key values" or "retrieve by content." As an example, if a business object handler receives a Customer business object with the verb RetrieveByContent and with the non-key attributes Name and City set to `Smith` and `San Diego`, the RetrieveByContent operation can attempt to retrieve a customer entity that matches the values of the Name and City attributes.

When the business object handler obtains a RetrieveByContent verb from the request business object, it must ensure that an existing application entity, whose type is indicated by the business object definition, is retrieved, as follows:

- For a flat business object, the RetrieveByContent verb indicates that the specified entity is retrieved by its non-key values. The verb operation returns a business object that contains the current values for the application entity.
- For a hierarchical business object, the RetrieveByContent verb indicates that one or more application entities (to match the entire business object) are retrieved by the non-key values of the top-level business object. The verb operation returns a business object in which all simple attributes of each business object in the hierarchy match the values of the corresponding entity attributes, and the number of individual business objects in each child business object array matches the number of child entities in the application.

This section provides the following information to help process a RetrieveByContent verb:

- "Implementation for a RetrieveByContent verb operation" on page 96
- "Outcome status for RetrieveByContent processing" on page 96

Note: You can modularize your business object handler so that each supported verb is handled in a separate Java method. If you follow this structure, a RetrieveByContent method handles processing for the RetrieveByContent verb.

Implementation for a RetrieveByContent verb operation

RetrieveByContent functions the same as the Retrieve verb except that it uses a subset of non-key values, instead of key values, to retrieve application data. In its most robust implementation, a top-level business object and its child business objects would independently support the RetrieveByContent verb. However, not all application APIs enable retrieve by non-key values for hierarchical business objects.

A more common implementation is to provide RetrieveByContent support only in the top-level business object. When a top-level business object supports retrieve by non-key values and this retrieve-by-content is successful, the RetrieveByContent operation can retrieve the keys for the entity matching the request business object. The verb operation can then perform a Retrieve operation to retrieve the complete business object.

You might want to specify which attributes are to be used in RetrieveByContent operations. To do this, you can implement attribute application-specific information to specify those attributes that will contain a value that is to be used in the RetrieveByContent operation or receive a value as a result of the operation.

Outcome status for RetrieveByContent processing

The RetrieveByContent operation should return one of the outcome-status values shown in Table 37..

Table 37. Possible outcome status for Java RetrieveByContent verb processing

RetrieveByContent condition	Java outcome status
If the RetrieveByContent operation finds a single entity that matches the query, it: • fills the entire business object hierarchy, including all child business objects; this business object is returned to the connector framework through the request business object parameter. • returns a "Value Changed" outcome status	VALCHANGE
If the IgnoreMissingChildObject connector property is True, the RetrieveByContent operation returns the "Value Changed" outcome status for the business object if it is successful in retrieving the top-level object, regardless of whether it is successful in retrieving its children.	VALCHANGE
If the RetrieveByContent operation finds multiple entries that match the query, it: • retrieves only the first occurrence of the match; this business object is returned to the connector framework through the request business object parameter. • fills a return-status descriptor with further information about the search • returns a status of "Multiple Hits" to notify the connector framework that there are additional records that match the specification	MULTIPLE_HITS

Table 37. Possible outcome status for Java RetrieveByContent verb processing (continued)

RetrieveByContent condition	Java outcome status
If the RetrieveByContent operation does *not* find matches for retrieve by non-key values, it: • fills a return-status descriptor containing additional information about the cause of the RetrieveByContent error • returns a "RetrieveByContent Failed" outcome status	RETRIEVEBYCONTENT_FAILED

Note: When the connector framework receives the VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 169."Sending the verb-processing response" on page 179.

Handling the Update verb

When the business object handler obtains an Update verb from the request business object, it must ensure that an existing application entity, whose type is indicated by the business object definition, is updated, as follows:

- For a flat business object, the Update verb indicates that the data in the specified entity must be modified as necessary until the application entity exactly matches the request business object.
- For a hierarchical business object, the Update verb indicates that updates the application entity must be updated to match the entire business object hierarchy. To do this, the connector might need to create, update, and delete application entities:
  - If child entities exist in the application, they are modified as needed.
  - Any child business objects contained in the hierarchical business object that do *not* have corresponding entities in the application are added to the application.
  - Any child entities that exist in the application but are *not* contained in the business object are deleted from the application.

Note: For a table-based application, the entire application entity must be updated in the application database, usually as a new row to the database table associated with the business object definition of the request business object.

This section provides the following information to help process an Update verb:

- "Standard processing for an Update verb"
- "Implications of business objects representing logical Delete events" on page 101
- "Outcome status for Update verb processing" on page 103

Note: You can modularize your business object handler so that each supported verb is handled in a separate Java method. If you follow this structure, an Update method handles processing for the Update verb.

Standard processing for an Update verb

The following steps outline the standard processing for an Update verb:

1. Create a new business object of the same type as the request business object. This new business object is the *response business object*, which will hold the retrieved copy of the request business object.

2. Retrieve a copy of the request business object from the application.

   Recursively retrieve the data for the entire entity from the application using the primary keys from the request business object:

   - For a flat business object, retrieve the single application entity.
   - For a hierarchical business object, use the Retrieve operation to descend into the application business object, expanding all paths in the business object hierarchy.

3. Place the retrieved data in the response business object. This response business object is now a representation of the current state of the entity in the application.

   The Update operation can now compare the two hierarchical business objects and update the application entity appropriately.

4. Update the simple attributes in the application entity to correspond to the top-level source business object.
5. Compare the response business object (created in step 2) with the request business object. Perform this comparison down to the lowest level of the business object hierarchy.

   Recursively update the children of the top-level business object following these rules:

   - If a child business object is present in *both* the response business object and the request business object, recursively update the child by performing the Update operation.
   - If a child business object is present in the request business object but *not* in the response business object, recursively create the child by performing the Create operation.
   - If a child business object is *not* present in the request business object but is present in the response business object, recursively delete the child using either the Delete operation (physical) or a logical delete, depending on the functionality of the connector and the application. For more information on logical deletes, see "Implications of business objects representing logical Delete events" on page 101.

   Note: Only the existence or non-existence of the child objects are compared, *not* the attributes of the child business objects.

If the connector's application supports logical delete, the connector recursively retrieves the complete business object hierarchy; then the Update operation sets status attributes and recursively updates the status of the children.

Note: The Update operation should fail if an application entity does *not* exist for any foreign key (Foreign Key is set to true) referenced in the request business object. The connector should verify that the foreign key is a valid key (it references an existing application entity). If the foreign key is invalid, the Update operation should return FAIL. A foreign key is assumed to be present in the application, and the connector should *never* try to create an application object marked as a foreign key.

Figure 34 shows a set of associated application entities that represent a customer in the application database. The entities contain customer, address, phone, and customer profile data. Note that the sample customer, Acme Construction, has no phone number in the database.

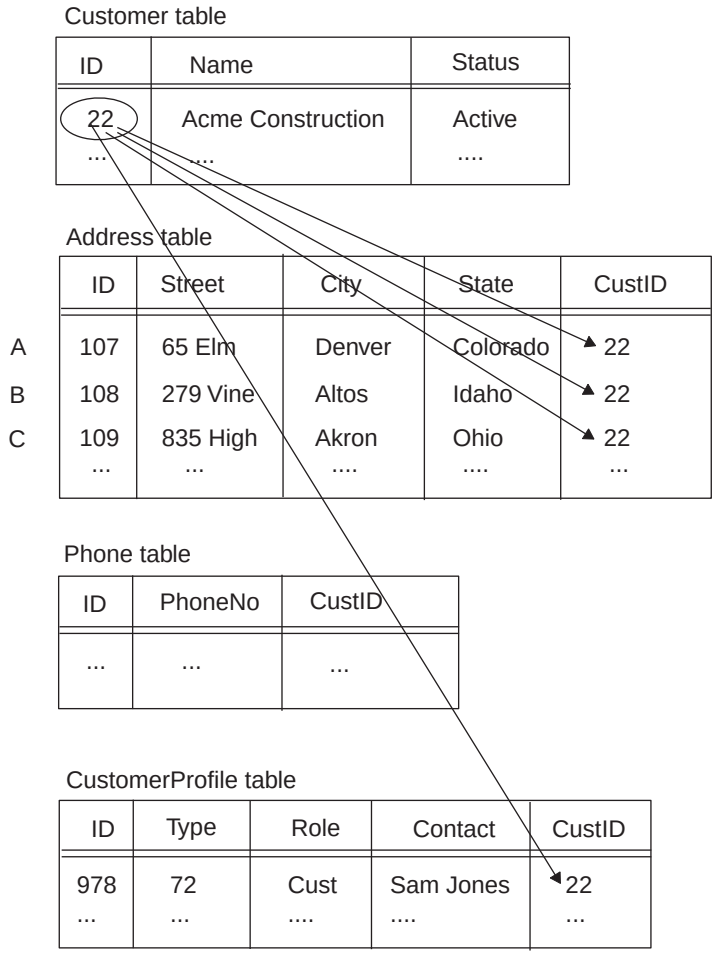

Figure 34. Customer entities before Update request

Assume that an integration broker sends an update request that consists of the request business object as shown in Figure 35..

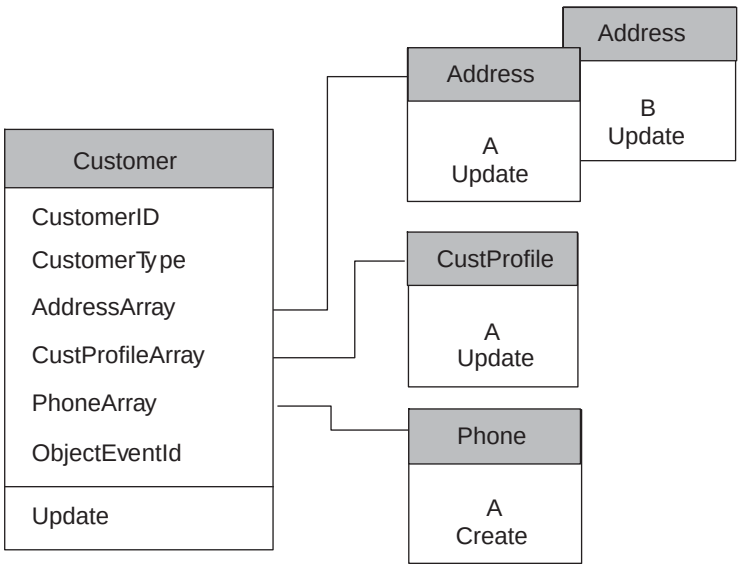

Figure 35. Customer request business object for an Update

This request business object indicates that the Acme Construction customer has undergone the changes listed in Table 38..

Table 38. Updates to Acme Construction in the Request business object

Update made to Acme Construction	Representation in request business object
Acquired a new phone number	The child business object for the PhoneArray attribute (Phone object A) has a Create verb.
Moved to new offices in Denver and Altos	Two child business objects (Address objects A and B) exist in the AddressArray attribute, each with an Update verb.
Closed the office in Akron	No child business object exists in the AddressArray attribute for the Akron address.
Changed the name of the contact person	The child business object for the CustProfileArray attribute (CustProfile object A) has an Update verb.

Your connector's task is to keep the application database for this destination application synchronized with the source application. Therefore, to respond to this request, the connector would need to perform the following tasks as part of its Update operation:

- Update any columns in `Customer` table that have updated values in the corresponding simple attributes of the Customer business object.
- Update the rows in the `Address` table that correspond to Address objects A and B. Update the columns in each of these rows with any new values from the corresponding simple attributes in the appropriate Address object. In this case, the `Street` column has changed for the Denver and Altos offices.
- Delete the row in the `Address` table that corresponds to the Akron address.
- Update the `Contact` column of the `CustomerProfile` table to the value of the corresponding simple attribute in the CustProfile object A business object.
- Create a row in the `Phone` table with column values from the simple attributes of the Phone object A business object. Make sure that the `CustID` column of this new row is created with the foreign-key value that identifies the appropriate `Customer` row (22).

Figure 36 shows the set of associated application entities that represent a customer after the Update operation has completed.

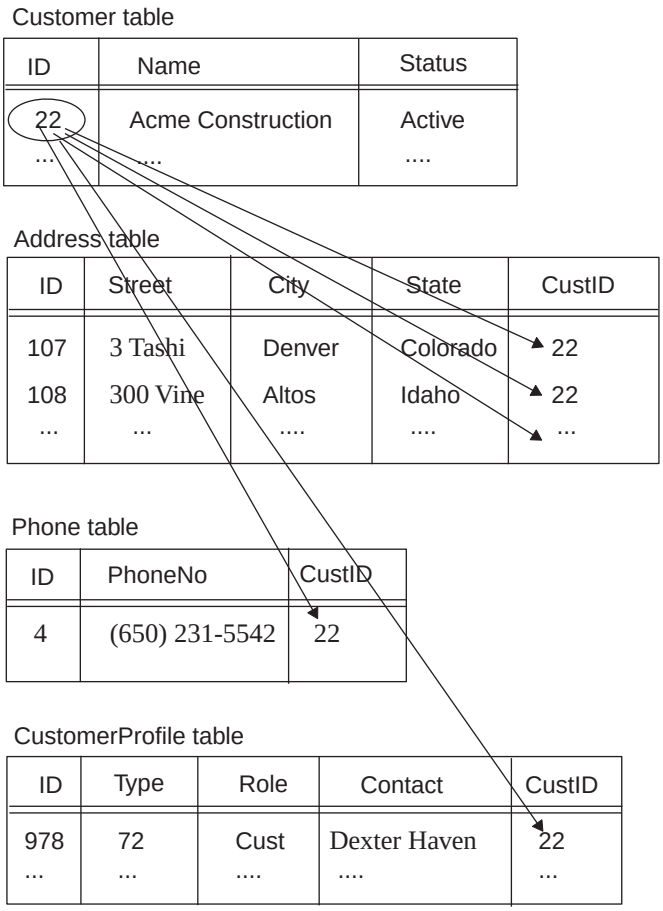

Figure 36. Customer entities after Update request

Implications of business objects representing logical Delete events

If your application supports physical delete, but an integration broker sends requests from a source application that supports only logical delete, you might need to handle a business object that represents a logical delete request. Connectors for applications that perform logical delete operations, where an entity is marked as deleted by updating a status value, should handle logical deletes in the Update method. A system view of this implementation is as follows:

- Events that represent the deletion of data in the source application should be sent as application-specific business objects with the Delete verb. Similarly, maps on the source application side should set the verb of generic business objects to Delete.
- On the destination side, maps for connectors supporting logical delete applications can transform Delete verbs in generic business objects to Update verbs in application-specific business objects. Business object attributes representing entity status values can be set to the inactive status.

In this way, a connector representing a logical delete application receives an application-specific business object with an Update verb and the status value marked appropriately.

For example, assume that a source application entity has been updated to look like the business object representation in Figure 37.. Components in the source application entity have been updated, created, and deleted.

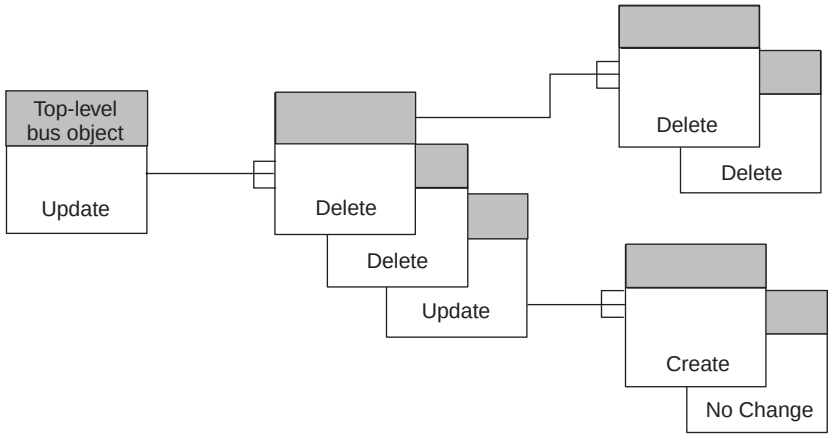

Figure 37. Updated entity in the source application

If the source application connector has implemented event notification as recommended in Chapter 5, "Event notification," on page 115, deleted child business objects are not present in the business object hierarchy, and the business object simply contains the updated and new child business objects.

An example of a business object representing an Update request might look like Figure 38.. In this figure, the parent object is set to update, and all entities that have been deleted are no longer present in the business object hierarchy.

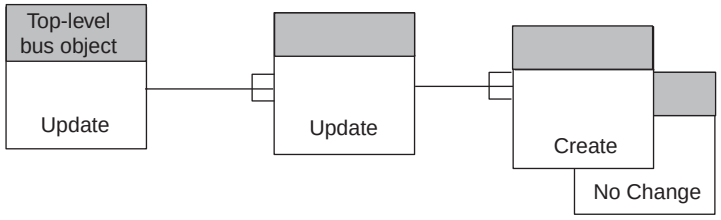

Figure 38. Update request business object from a physical-delete connector

In this case, the connector compares the source and destination business objects and deletes the entities that are not present in the source business object.

However, if the source application supports logical delete, the source connector might send a business object with deletes tagged as updates and status attribute values set to an inactive value. This business object might look like Figure 39,, where updates that are delete operations are identified by "[D]".

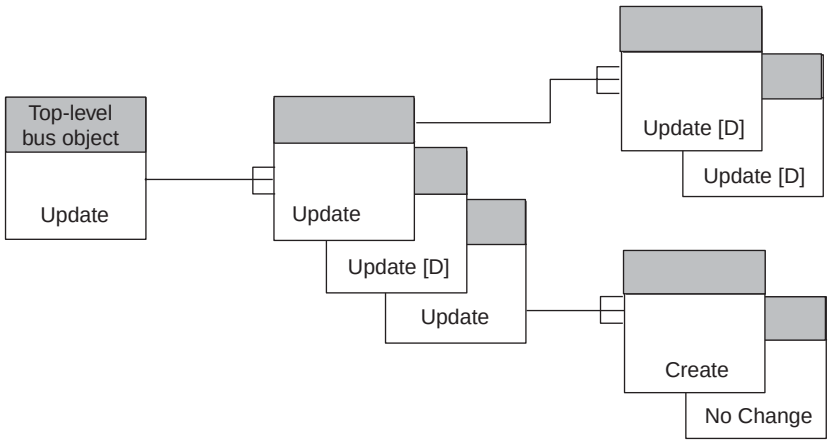

Figure 39. Update request business object from a logical-delete connector

There are several ways to handle a source business object that represents a logical delete request:

- Implement mapping to examine the status of child business objects. If the status of a particular child business object is inactive, the business object can be removed in mapping.
- Implement the Update operation to determine whether an update operation is actually a delete operation. In a logical delete source application, an entity may be marked as active or inactive by a status value. In the source's application-specific business objects, the status value is usually an attribute. Although entities in an application that supports physical delete might not include status information, you can extend your application-specific business objects to include status information.
- Extend a business object by adding an additional status attribute or by overloading an existing attribute with a status value. When the Update operation receives a request, it can check the status attribute. If it is set to the inactive value, the operation is really a delete. The Update operation can then set the business object verb to Delete and call the Delete operation to handle deleted child business objects.

Outcome status for Update verb processing

The Update operation should return one of the outcome-status values shown in Table 39..

Table 39. possible outcome status for Java Update verb processing

Update condition	Java outcome status
If the application entity exists, the Update operation: • modifies the data in the application entity • returns a "Success" outcome status	SUCCEED
If a row or entity does not exist, the Update operation: • creates the application entity • returns the "Value Changed" outcome status to indicate that the connector has changed the business object	VALCHANGE

Table 39. possible outcome status for Java Update verb processing (continued)

Update condition	Java outcome status
If the Update operation is unable to create the application entity, it: • fills a return-status descriptor with information about the cause of the update error • returns a "Fail" outcome status	FAIL
If any object identified as a foreign key is missing from the application, the Update operation: • fills a return-status descriptor with information about the cause of the update error • returns a "Fail" outcome status	FAIL

Note: When the connector framework receives the VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 169."Sending the verb-processing response" on page 179.

Handling the Delete verb

For a delete, an application might support either of the implementations shown in Table 40..

Table 40. Delete Implementations

Delete implementation	Description	Verb-processing support
Physical delete	Physically removes the specified application entity.	Delete operation
Logical delete	Does not actually remove the entity; instead, it marks it with a special "deleted" status.	Update operation

Note: If the application does not allow *any* type of delete operation, the connector can return a "Fail" outcome status.

The Delete operation, discussed in this section, performs a true physical deletion of data in the application. Connectors for applications that perform logical delete operations should handle logical deletes in the Update operation. For more information, see "Implications of business objects representing logical Delete events" on page 101.

When the business object handler obtains a Delete verb from the request business object, it must ensure that a physical delete is performed; that is, the application deletes the application entity whose type is indicated by the business object definition, as follows:

- For a flat business object, the Delete verb indicates that the specified entity must be deleted.
- For a hierarchical business object, the Delete verb indicates that the top-level business object must be deleted. Depending on the application policies, the it might delete associated entities representing child business objects.

Note: For a table-based application, the entire application entity must be deleted from the application database, usually deleting a row in one or more database tables.

This section provides the following information to help process a Delete verb:

- "Standard processing for a Delete verb"
- "Outcome status for Delete verb processing"

Note: You can modularize your business object handler so that each supported verb is handled in a separate Java method. If you follow this structure, a Delete method handles processing for the Delete verb.

Standard processing for a Delete verb

The following steps outline the standard processing for a Delete verb:

1. Perform a recursive retrieve on the request business object to get all data in the application that is associated with the top-level business object.
2. Perform a recursive delete on the entities represented by the request business object, starting from the lowest level entities and ascending to the top-level entity.

Note: Delete operations might be limited by application functionality. For example, cascading deletes might not always be the desired operation. If you are using an application API, it might automatically complete the delete operation appropriately. If you are *not* using an application API, you might need to determine whether the connector should delete child entities in the application. If a child entity is referenced by other entities, it might not be appropriate to delete it.

Outcome status for Delete verb processing

The Delete operation should return one of the outcome-status values shown in Table 41..

Table 41. Possible outcome status for Java Delete verb processing

Delete condition	Java outcome status
InterChange Server only: In most cases, the connector returns a "Value Changed" outcome status to enable the system to clean up the relationship tables after a delete operation.	VALCHANGE
All integration brokers: If the Delete operation is unsuccessful, it: • fills a return-status descriptor with additional information about the cause of the delete error • returns a "Fail" outcome status	FAIL

Note: When the connector framework receives the VALCHANGE outcome status, it includes a business object in its response to InterChange Server. For more information, see "Sending the verb-processing response" on page 169."Sending the verb-processing response" on page 179.

Handling the Exists verb

When the business object handler obtains an Exists verb from the request business object, it must determine whether an application entity, whose type is indicated by the business object definition, exists. This operation enables an integration broker to verify that an entity exists before the integration broker performs an operation on the entity. As an example, assume that a customer site wants to synchronize Order, Customer, and Item entities in the source and destination applications. Before synchronizing an order, the user wants to ensure that the customer entity

referenced by the Order business object already exists in the destination application database. In addition, the user wants to ensure that each Item entity referenced by the OrderLineItem child business objects also exists in the destination application.

Note: For a table-based application, the Exists method checks for the existence of an entity in an application database, usually checking for a row in a database table.

The user can configure the integration broker to call the connector with a Customer business object that has the Exists verb and the primary keys set. In this way, the integration broker can verify that the customer already exists in the application. Similarly, the user can configure the integration broker to call the connector with referenced Item business objects that have the Exists verb and primary keys set. The user might decide to halt the synchronization of the Order if the verification of the existence of the application entities fails.

This section provides the following information to help implement an Exists verb:

- "Standard processing for an Exists verb"
- "Outcome status for Exists verb processing"

Note: You can modularize your business object handler so that each supported verb is handled in a separate Java method. If you follow this structure, an Exists method handles processing for the Exists verb.

Standard processing for an Exists verb

The standard behavior of the Exists method is to query the application database for the existence of a top-level business object.

Outcome status for Exists verb processing

The Exists operation should return one of the outcome-status values shown in Table 42..

Table 42. Possible outcome status for Java Exists verb processing

Exists condition	Java outcome status
If the application entity exists, the Exists operation returns "Success".	`SUCCEED`
If the Exists operation is unsuccessful in retrieving the top-level object, it: • fills a return-status descriptor • returns a "Fail" outcome status	`FAIL`

Processing business objects

A business object handler's role is to deconstruct a request business object, process the request, and perform the requested operation in the application. To do this, a business object handler extracts verb and attribute information from the request business object and generates an API call, SQL statement, or other type of application interaction to perform the operation.

Basic business object processing involves extracting metadata from the business object's application-specific information (if it exists) and accessing the attribute values. The actions to take on the attribute value depend on whether the business

object is flat or hierarchical. This section provides an overview on how a business object handler can process the following kinds of business objects:

- "Processing flat business objects"
- "Processing hierarchical business objects" on page 109

Processing flat business objects

This section provides the following information on how to process flat business objects:

- "Representing a flat business object"
- "Accessing simple attributes" on page 108

Representing a flat business object

If a business object does *not* contain any other business objects (called child business objects), it is called a *flat business object*. All the attributes in a flat business object are *simple attribute*; that is, each attribute contains an actual value, not a reference to another business object.

Suppose you have to perform verb processing on an example business object named `Customer`. This business object represents a single database table in a sample table-based application. The database table is named `customer`, and it contains customer data. Figure 40 shows the `Customer` business object definition and the corresponding `customer` table in the application.

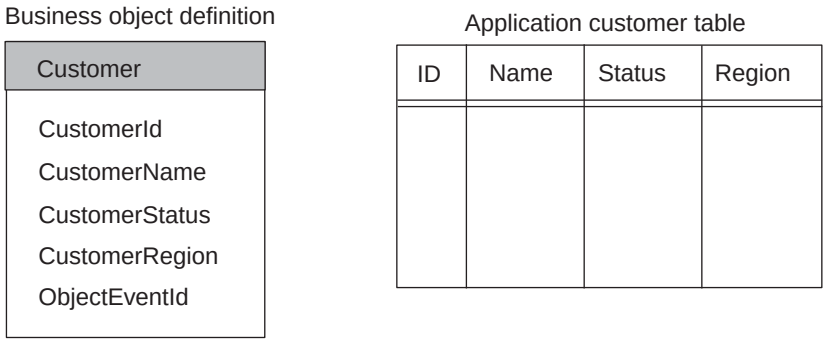

Figure 40. A Flat business object and corresponding application table

As Figure 40 shows, the example `Customer` business object has four simple attributes: `CustomerId`, `CustomerName`, `CustomerStatus`, and `CustomerRegion`. These attributes correspond to columns in the `customer` table. The business object also includes the required `ObjectEventId` attribute.

Note: The `ObjectEventId` attribute is used by the IBM WebSphere business integration system and does *not* correspond to a column in an application table. This attribute is automatically added to business objects by Business Object Designer.

Figure 41 shows an expanded business object definition and an instance of the business object. The business object definition contains the business object name, and the attribute name, properties, and application-specific information. The business object instance contains only the business object name, the active verb, and the attribute names and values.

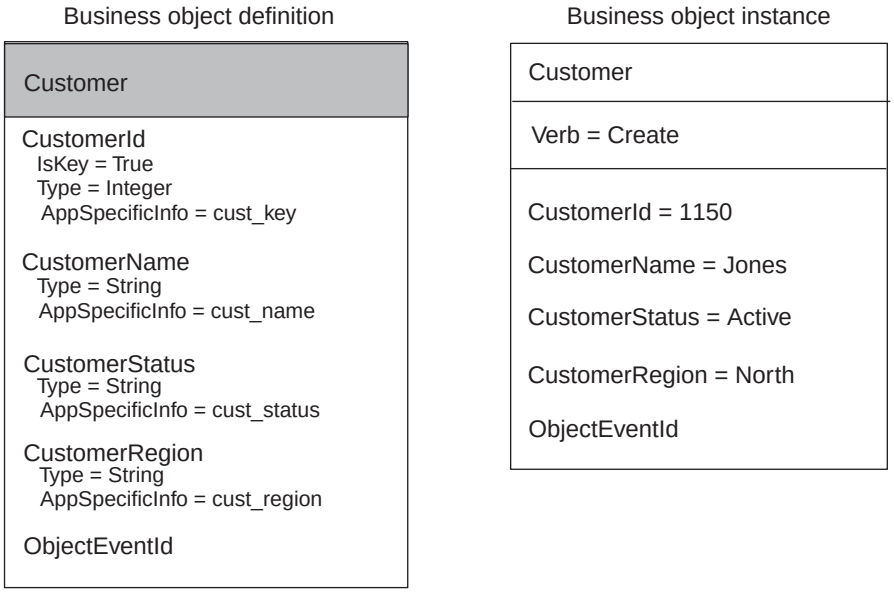

Figure 41. A flat business object with application-specific information

Accessing simple attributes

After the verb operation has accessed information it needs within the business object definition, it often needs to access information about attributes. Attribute properties include the cardinality, key or foreign key designation, and maximum length. For example, the example Create method needs to obtain the attribute's application-specific information. A connector business object handler typically uses the attribute properties to decide how to process the attribute value.

Figure 42 illustrates business object attribute properties of the `CustomerId` attribute from the business object in Figure 41..

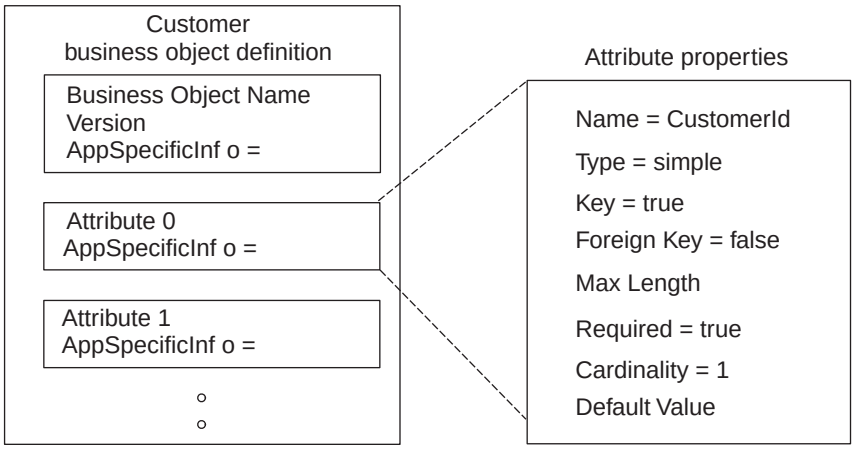

Figure 42. Business object attribute properties

Each attribute has a zero-based integer index (ordinal position) within the business object definition. For example, as Figure 42 shows, the `CustomerId` attribute would be accessed with an ordinal position of zero (`0`), the `CustomerName` attribute with an

ordinal position of one (1), and so on. The Java connector library provides access to an attribute through its name or ordinal position.

For the business object handler that handles the flat `Customer` business object, deconstructing a business object includes the following steps:

1. Extract the table and column names from the application-specific information in the business object definition.
2. Extract the values of the attributes from the business object instance.

As Figure 41 shows, the `Customer` business object definition is designed for a metadata-driven connector. Its business object definition includes application-specific information that the verb operation uses to locate the application entity upon which to operation. The application-specific information is designed as shown in Table 43..

Table 43. Application-specific information for a table-based application

Application-specific information	Purpose
Business object definition	The name of application database table associated with this business object
Attribute	The name of the application table's column associated with this attribute

Note: Application-specific information is also used to store information on foreign keys and other kinds of relationships between entities in the application database. A metadata-driven connector can use this information to build a SQL statement or an application API call.

Processing hierarchical business objects

Business objects are hierarchical: parent business objects can contain child business objects, which can in turn contain child business objects, and so on. A hierarchical business object is composed of a *top-level business object*, which is the business object at the very top of the hierarchy, and *child business objects*, which are all business objects under the top-level business object. A child business object is contained in a parent object as an attribute.

This section provides the following information on how to process hierarchical business objects:

- "Representing Top-Level and Child Business Objects"
- "Accessing child business objects" on page 111

Representing Top-Level and Child Business Objects

If a top-level business object has child business objects, it is the parent of its children. Similarly, if a child business object has children, it is the parent of its children. The parent/child terminology describes the relationships between business objects, and it may also be used to describe the relationship between application entities.

There are two types of containment relationships between parent and child business objects:

- Cardinality 1 containment—the attribute contains a single child business object.
- Cardinality n containment—the attribute contains several child business objects in a structure called a *business object array*.

Figure 43 shows a typical hierarchical business object. The top-level business object has both cardinality 1 and cardinality n relationships with child business objects.

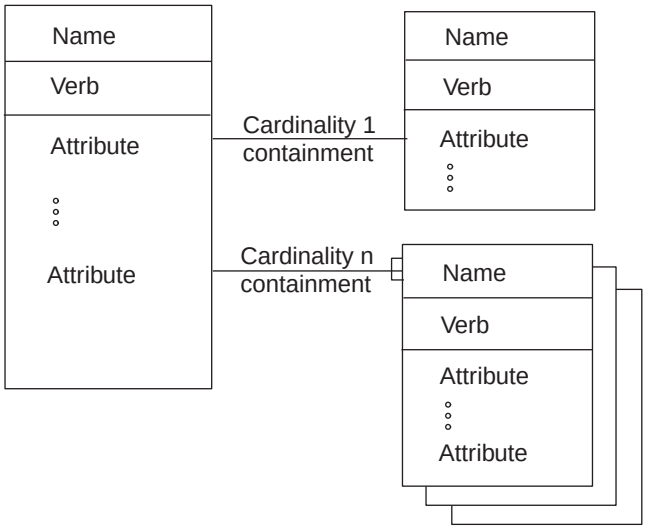

Figure 43. Hierarchical business object

In a typical table-based application, relationships between entities are represented by primary keys and foreign keys in the database, where the parent entity contains the primary keys and the child entity contains the foreign keys. An hierarchical business object can be organized in a similar way:

- In a cardinality 1 type (single cardinality) of relationship, each parent business object relates to a single child business object.

  The child business object typically contains one or more foreign keys whose values are the same as the corresponding primary keys in the parent business object. Although applications might structure the relationships between entities in different ways, a single cardinality relationship for an application that uses foreign keys might be represented as shown in Figure 44..
- In a cardinality n type (multiple cardinality) relationship, each parent business object can relate to zero or more child business objects in an array of child business objects.

  Each child business object within the array contains foreign key attributes whose values are the same as the corresponding values in the primary key attributes of the parent business object. A multiple cardinality relationship might be represented as shown in Figure 45..

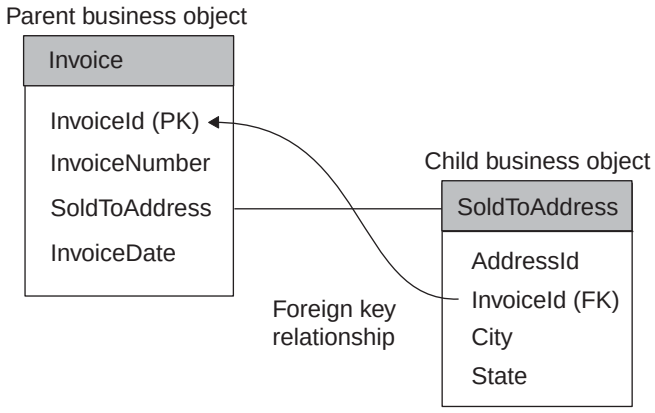

Figure 44. Business objects with single cardinality

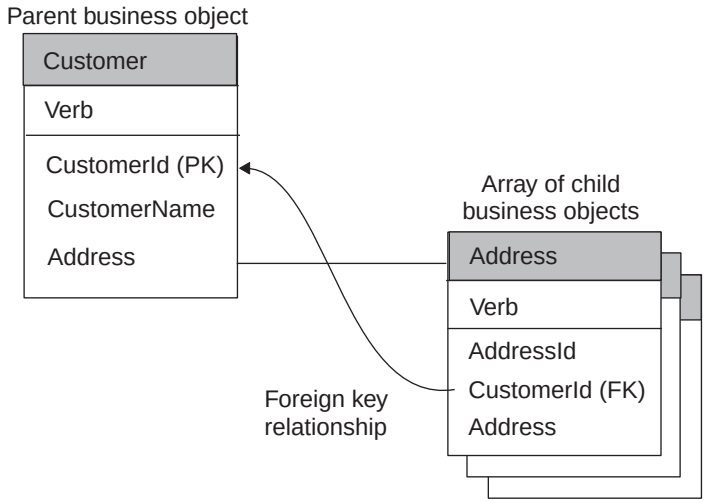

Figure 45. Business objects with multiple cardinality

Note: In Figure 44 and Figure 45,, the string "PK" appears next to an attribute that serves as a primary key in the business object. The string"FK" appears next to an attribute that serves as a foreign key.

Accessing child business objects

As part of its verb processing, the `doVerbFor()` method needs to handle any hierarchical business objects. To process a hierarchical business object, the `doVerbFor()` method takes the same basic steps as it does to process a flat business object: it obtains any application-specific information and then accesses the attribute. However, if the attribute contains a child business object, `doVerbFor()` must take the following steps to access the child business object:

1. Determine whether the attribute type is type `OBJECT` by calling the `isObjectType()` method.

   The `OBJECT` type indicates that the attribute is a complex attribute; that is, it contains a business object rather than a simple value. The `OBJECT` attribute-type constant is defined in the CWConnectorAttrType class. The `isObjectType()` method returns True if an attribute is complex; that is, if it contains a business object.
2. When the `doVerbFor()` method finds an attribute contains a business object, it checks the cardinality of the attribute using `isMultipleCard()`.

If the attribute has single cardinality (cardinality 1), the method can perform the requested operation on the child. One way to perform an operation on a child business object is to recursively call `doVerbFor()` or a verb method on the child object. However, such a recursive call assumes that the child business object is set as follows:

- If the verb on a child business object is set, the method should perform the specified operation.
- If the verb on the child business object is *not* set, the verb method should set the verb in the child business object to the verb in the top-level business object before calling another method on the child.

If an attribute has multiple cardinality (cardinality n), the attribute contains an array of child business objects. In this case, the connector must access the contents of the array before it can process individual child business objects. From the array, the `doVerbFor()` method can access individual business objects:

- To access individual business objects, the method can get the number of child business objects in the array with the `getObjectCount()` method and then iterate through the objects.
- To get an individual child business object, the method can obtain the business object at one element of the array.

Once the `doVerbFor()` method has access to a child business object, it can recursively process the child as needed.

Note: A connector should never create arrays for child business objects. An array is always associated with a business object definition when cardinality is n.

When a connector a request business object, the business object includes all its arrays even though some or all of the arrays might be empty. If an array contains no child business objects, it is an array of size 0.

You might want to modularize your verb operation so that the main verb method calls a submethod to process child objects. For a business object such as the one shown in Figure 46,, a Create method might first create the application entity for the parent Customer business object, and then call the submethod to traverse the parent business object to find attributes referring to contained business objects.

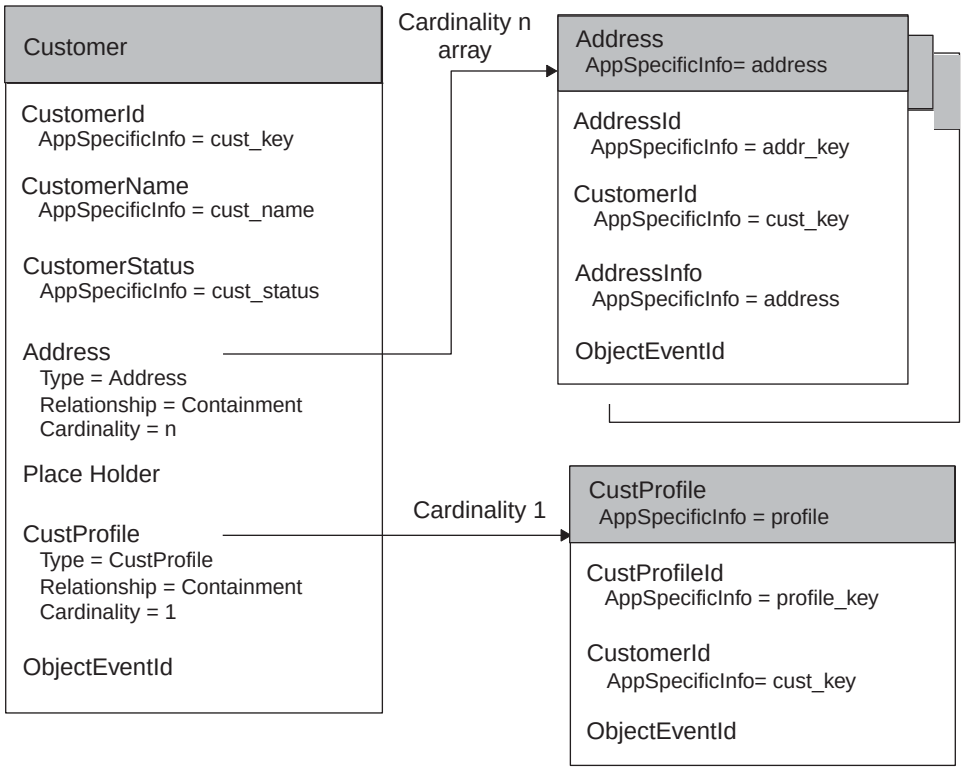

Figure 46. Example of a hierarchical business object definition

When the submethod finds an attribute that is an `OBJECT` type, it can process the attribute as needed. For example, the submethod processes the `Address` attribute by retrieving each child business object in the `Address` array and recursively calling `doCreate()`. One by one, the main method creates each `address` entity in the database until all `Address` children in the array are processed. Finally, the submethod processes the single cardinality `CustProfile` business object.

For more information about how to access a child business object, see "Accessing child business objects" on page 173."Accessing child business objects:" on page 183.

Indicating the connector response

Before the `doVerbFor()` method exits, it must prepare the response it sends back to the connector framework. This response indicates the success (or lack thereof) of the verb processing. The connector framework, which has invoked `doVerbFor()`, uses this information to determine its next action and to build the response it returns to the integration broker.

The `doVerbFor()` method can provide the response information in Table 44 to the connector framework.

Table 44. Response information from the doVerbFor() method

Response information	How the response is returned
Outcome status	Integer return code of `doVerbFor()`

Table 44. Response information from the doVerbFor() method (continued)

Response information	How the response is returned
Return-status descriptor	Return-status descriptor that was passed in as an argument—Connector framework passes in an empty return-status descriptor as an argument to `doVerbFor()`. The method can update this descriptor with a message and status value to provide informational, warning, or error status.
Response business object	Request business object that was passed in as an argument—Connector framework passes in the request business object as an argument to `doVerbFor()`. The method can update this request business object with attribute values to provide a response business object.

For information on how to send this response information for a Java connector, see "Sending the verb-processing response" on page 169."Sending the verb-processing response" on page 179.

Handling loss of connection to the application

Each time the connector framework calls the connector application-specific component, the application-specific code validates that the connection with the application is still open. For a business object handler, this check should be done in either the `doVerbFor()` method or in each verb method.

If the connection has been lost, the `doVerbFor()` method should log a fatal error message so that email notification is triggered if the `LogAtInterchangeEnd` connector configuration property is set to True. The method should also return a APPRESPONSETIMEOUT outcome status to inform the connector controller that the application is not responding. When this occurs, the process in which the connector runs is stopped. A system administrator must fix the problem with the application and restart the connector to continue processing of business object requests.

For more information, see "Verifying the connection before processing the verb" on page 158."Verifying the connection before processing the verb" on page 168.

Chapter 5. Event notification

This chapter presents information on how to provide event notification in a connector. *Event notification* implements a mechanism to interact with an application to detect changes made to application business entities. This chapter provides the following information about how to implement an event-notification mechanism:

- "Overview of an event-notification mechanism"
- "Implementing an event store for the application" on page 116
- "Implementing event detection" on page 121
- "Implementing event retrieval" on page 126
- "Implementing the poll method" on page 128
- "Special considerations for event processing" on page 132

Note: For an introduction to event notification, see "Event notification" on page 22..

Overview of an event-notification mechanism

An *event-notification mechanism* enables a connector to determine when an entity within an application changes. Implementation of an event-notification mechanism is a three-stage process, as Table 45 shows.

Table 45. Stages of an event-notification mechanism

Stage of event-notification mechanism	For more information
Create an *event store* that the application uses to hold notifications of events that have changed application business entities.	"Implementing an event store for the application" on page 116
Implement an **event detection** mechanism within the application. Event detection notices a change in an application entity and writes an **event record** containing information about the change to an event store in the application.	"Implementing event detection" on page 121
Implement an **event retrieval** mechanism (such as a polling mechanism) within the connector to retrieve events from the event store and take the appropriate action to notify other applications.	"Implementing an event store for the application" on page 116

Note: For design considerations for an event-notification mechanism, see "Event notification" on page 22..

In many cases, an application must be configured or modified before the connector can use the event-notification mechanism. Typically, this application configuration occurs as part of the installation of the connector's application-specific component. Modifications to the application might include setting up a user account in the application, creating an event store and event table in the application database, inserting stored procedures in the database, or setting up an inbox. If the application generates event records, it might be necessary to configure the text of the event records.

The connector might also need to be configured to use the event-notification mechanism. For example, a system administrator might need to set connector-specific configuration properties to the names of the event store and event table.

Implementing an event store for the application

An *event store* is a persistent cache in the application where event records are saved until the connector can process them. The event store might be a database table, application event queue, email inbox, or any type of persistent store. If the connector is not operational, a persistent event store enables the application to detect and save event records until the connector becomes operational.

This section provides the following information about an event store:

- "Standard contents of an event record"
- "Possible implementations of an event store" on page 118

Standard contents of an event record

Event records must encapsulate everything a connector needs to process an event. Each event record should include enough information that the connector poll method can retrieve the event data and build a business object that represents the event.

Note: Although different event retrieval mechanisms might exist, this section describes event records in the context of the most common mechanism, polling.

If the application provides an event detection mechanism that writes event records to an event store, the event record should provide discrete detail on the object and verb. If the application does not provide sufficient detail, it might be possible to configure it to provide this level of detail.

Table 46 lists the standard elements for event records. The sections that follow include more information on certain fields.

Table 46. Standard elements of an event record

Element	Description	For more information
Event identifier (ID)	A unique identifier for the event.	"Event identifier" on page 117
Business object name	The name of the business object definition as it appears in the repository.	"Business object name" on page 117
Verb	The name of the verb, such as Create, Update, or Delete.	"Event verb" on page 117
Object key	The primary key for the application entity.	"Object key" on page 117
Priority	The priority of the event in the range 0 - n, where 0 is the highest priority.	"Processing events by event priority" on page 131
Timestamp	The time at which the application generated the event.	None.
Status	The status of the event. This is used for archiving events.	"Event status" on page 118
Description	A text string describing the event.	None
Connector identifier (ID)	An identifier for the connector that will process the event.	"Event distribution" on page 131

Note: A minimal set of information in an event record includes the event ID, business object name, verb, and object key. You may also want to set a priority for an event so that if large numbers of events are queued in the event store, the connector can select events in order of priority.

Business object name

You can use the name of the business object definition to check for event subscriptions. Note that the event record should specify the *exact* name of the business object definition, such as `SAP_Customer` rather than `Customer`.

Event verb

The verb represents the kind of event that occurred in the application, such as Create, Update, or Delete. You can use the verb to check for event subscriptions.

Note: Events that represent deletion of application data should generate event records with the Delete verb. This is true even for logical delete operations, where the delete is an update of a status value to inactive. For more information, see "Processing Delete events" on page 132.

The verb that the connector sets in the business object should be same verb that was specified in the event record.

Object key

The entity's object key enables the connector to retrieve the full set of entity data if the object has subscribing events.

Note: The only data from the application entity that event records should include are the business object name, active verb, and object key. Storing additional entity data in the event store requires memory and processing time that might be unneeded if no subscriptions exist for the event.

The object key column must use name/value pairs to set data in the event record. For example, if `ContractId` is the name of an attribute in the business object, the object key field in the event record would be:

```
ContractId=45381
```

Depending on the application, the object key may be a concatenation of several fields. Therefore, the connector should support multiple name/value pairs that are separated by a delimiter, for example ContractId=45381:HeaderId=321. The delimiter should be configurable as set by the `PollAttributeDelimiter` connector configuration property. The default value for the delimiter is a colon (:).

Event identifier

Each event must have a unique identifier. This identifier can be an number generated by the application or a number generated by a scheme that your connector uses. As an example of an event ID numbering scheme, the event may generate a sequential identifier, such as 00123, to which the connector adds its name. The resulting object event ID is `ConnectorName`_00123. Another technique might be to generate a timestamp, resulting in an identifier such as *`ConnectorName`*_06139833001001.

Your connector can optionally store the event ID in the `ObjectEventId` attribute in a business object. The `ObjectEventId` attribute is a unique value that identifies each event in the IBM WebSphere business integration system. Because this attribute is required, the connector framework generates a value for it if the application-specific connector does not provide a value. If no values for

ObjectEventIds are provided for hierarchical business objects, the connector framework generates values for the parent business object and for each child. If the connector generates ObjectEventId values for hierarchical business objects, each value must be unique across all business objects in the hierarchy regardless of level.

Event status

A Java connector should use the event-status constants, which are defined in CWConnectorEventStatusConstants class. Table 47 lists the event-status constants.

Table 47. Event-status values for a Java connector

Event-status constant	Description
READY_FOR_POLL	Ready for poll
SUCCESS	Sent to the integration broker
UNSUBSCRIBED	No subscriptions for event
IN_PROGRESS	Event is in progress
ERROR_PROCESSING_EVENT	Error in processing the event. A description of the error can be appended to the event description in the event record.
ERROR_POSTING_EVENT	Error in sending the event to the integration broker. A description of the error can be appended to the event description in the event record.
ERROR_OBJECT_NOT_FOUND	Error in finding the event in the application database

Possible implementations of an event store

The application might use any of the following as the event store:

- "Event inbox"
- "Event table" on page 119
- "Email" on page 120
- "Flat files" on page 121

Note: Some applications might provide multiple ways of keeping track of changes to application entities. For example, an application might provide workflow for some database tables and user exits for other tables. If this is the case, you may have to piece together an event notification mechanism that handles events in one way for some business objects and another way for other business objects.

Event inbox

Some applications have a built-in inbox mechanism. This inbox mechanism can be used to transfer information about application events to the connector, as follows:

- Event detection—you might need to identify the entities and events that trigger entries in the inbox.
- Event retrieval—the connector's application-specific component can retrieve the entries. If an API is available that provides interfaces to access the inbox, the application-specific component can use this API.

Figure 47 illustrates this interaction.

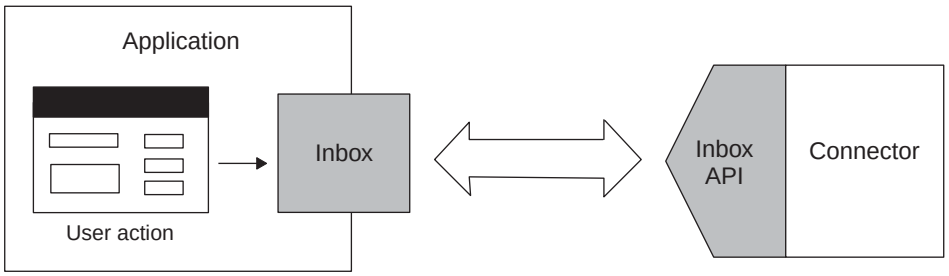

Figure 47. An event inbox as an event store

Event table

An application can use its application database to store event information. It can create a special *event table* in this database to use as the event store for event records. This table is created during the installation of the connector. With an event table as an event store:

- Event detection—when an event of interest to the connector occurs, the application places an event record in the event table.
- Event retrieval—the connector application-specific component polls the event table periodically and processes any events. Applications often provide database (DB) APIs that enable the connector to gain access to the contents of the event table.

Figure 48 illustrates this interaction.

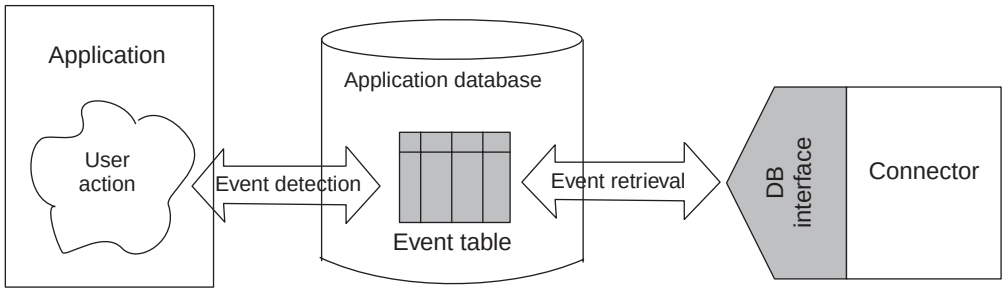

Figure 48. An event table as an event store

Note: Avoid full table scans of existing application tables as a way of determining whether application tables have changed. The recommended approach is to populate an event table with event information and poll the event table.

If your connector supports archiving of events, you can also create an archive table in the application database to hold the archived events. Table 48 shows a recommended schema for event and archive tables. You can extend this schema as needed for your application.

Table 48. Recommended schema for event and archive tables

Column name	Type	Description
`event_id`	Use appropriate type for database	The unique key for the event. System constraints determine format.
`object_name`	Char 80	Complete name of the business object.
`object_verb`	Char 80	Event verb.

Table 48. Recommended schema for event and archive tables (continued)

Column name	Type	Description
`object_key`	Char 80	The primary key of the object.
`event_priority`	Integer	The priority of the event, where 0 is the highest priority.
`event_time`	DateTime	The timestamp for the event (time at which the event occurred).
`event_processed`	DateTime	For the archive table only. The time at which the event was handed to the connector framework.
`event_status`	Integer	For possible status values, see "Event status" on page 118.
`event_description`	Char 255	Event description or error string
`connector_id`	Integer	Id for the connector (if applicable)

Email

You can use an email system as an event store:

- Event detection—the application sends an email message to a mailbox when an application event occurs.
- Event retrieval—the connector's application-specific component checks the mailbox and retrieves the event message.

Figure 49 illustrates this interaction.

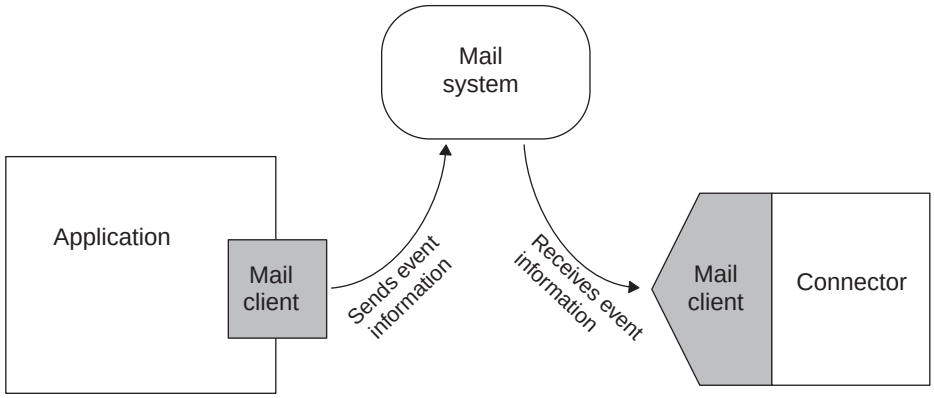

Figure 49. A mailbox as an event store

For an email-based event store, the mailbox used for a connector must be configurable, and the actual name of the inbox used should reflect its usage. The following list specifies the format and recommended names for fields in event messages.

- Message attributes – Email messages usually have certain attributes, such as a creation date and time, and a priority. You may be able to use these attributes in the event notification mechanism. For example, you may be able to use the date and time attributes to represent the date and time at which the event occurred.
- Subject – The subject of an event message might have the following format. In this example, fields are separated by spaces for human-readability, but connectors can use a different field delimiter.

  `object_name object_verb event_id`

  The *event_id* is the unique key for the event. Depending on the application, the *event_id* key may or may not be included in the mail message. The *event_id* can

be derived from a combination of the connector name, business object name, and either the message timestamp or the system time.

- Body – The body of an event message might contain a sequence of key/value pairs separated by delimiters. These key/value pairs are the elements of the object key. For example, if a particular customer and address are uniquely identified by the combination of CustomerId and AddrSeqNum, the body of the mail message might look like this:

  ```
  CustomerId 34225
  AddrSeqNum 2
  ```

  The body of the event message can be a list of attribute names for the business object, and the values that should be inserted into those attributes.

Flat files

If no other event detection mechanism is available, it might be possible to set up an event store using flat files. With this type of event store:

- Event detection—the event detection mechanism in the application writes event records to a file.
- Event retrieval—the connector's application-specific component locates the file and reads the event information.

If the file is not directly accessible by the connector (if, for example, it was generated on a mainframe system), the file must be transferred to a location that the connector can access. One way of transferring files is to use File Transfer Protocol (FTP). This can be done either internally in the connector or using an external tool to copy the file from one location to another. There are other ways to transfer information between files; the approach that you choose depends on your application and connector.

Figure 50 illustrates event detection and retrieval using flat files. In this example, FTP is used to transfer the event information to a location accessible by the connector.

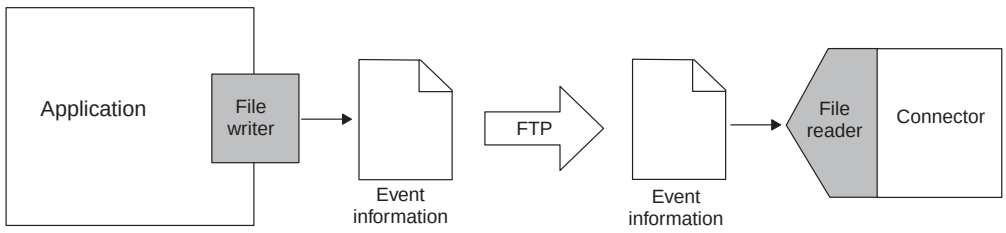

Figure 50. Retrieving event records from flat files

Implementing event detection

For most connectors, the application must be configured to implement the event detection mechanism. A system administrator does this as part of the connector installation. Once the application has been configured, it can detect entity changes and write event records to the event store. The information is then picked up by the connector and processed. In this way, an event notification mechanism is implemented in both the application and the connector.

This section provides the following information about event detection:

- "Event detection mechanisms" on page 122
- "Event detection: standard behavior" on page 125

Event detection mechanisms

Events can be triggered by user actions in the application, by batch processes that add or modify application data, or by database administrator actions. When an event detection mechanism is set up in an application and an application event associated with a business object occurs, the application must detect the event and write it to the event store.

Event detection mechanisms are application dependent. Some applications provide an event detection mechanism for use by clients such as connectors. The event detection mechanism may include an event store and a defined way of inserting information about application changes into the event store. For example, one type of implementation uses an event message box, where the application sends a message every time it processes an event in which the connector is interested. The connector's application-specific component periodically polls the message box for new event messages.

Other applications have no built-in event detection mechanism but have other ways of providing information on changes to application entities. If an application does not provide an event detection mechanism, you must use whatever mechanism is available to extract information on entity changes for the connector. For example, you may be able to implement database triggers, use user exits to call out to a program that writes to an event store, or extract information on application changes from flat files.

Note: Although the way in which events are generated can vary significantly from application to application, certain aspects of an event notification mechanism should be consistent across all types of applications. For example, all types of event detection mechanisms should create event records that have similar contents.

Three common ways in which events are detected and written to an event store are discussed in the following sections:

- "Form events"
- "Workflow" on page 123
- "Database triggers" on page 124

Form events

Some form-based applications provide form events that are executed when a special user action occurs. To set up event detection in this way, you must create a script that executes when a particular type of event occurs. When a user opens a form and performs an action that has an associated script, the script places event records in the event store.

In most cases, form events are integrated in application business processes and therefore support application business logic. However, only application events that are triggered by user actions are detected; if the application database is updated directly in other ways, such as by a batch process, these events are not detected.

Figure 51 shows a form-based event detection mechanism. When a user enters a new customer on the Customer form and clicks OK, a script generates an event record and places it in the event store.

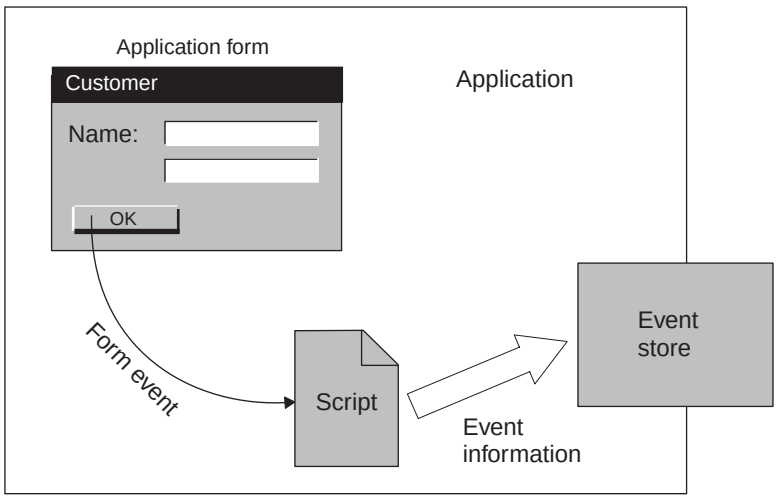

Figure 51. Form-based event detection

Workflow

Some applications use an internal workflow system to keep track of their business processes. You may be able to use the workflow system to generate events for event detection.

For example, you may be able to define a workflow process that inserts an entry in an event store when a particular operation occurs. Alternatively, the event detection mechanism might be able to intercept information from a workflow process and use the information to place an event record in the event store. In designing a workflow-based event detection mechanism, you need to determine at what point in the workflow an event record should be written to the event store and then use the available application mechanism to generate the event record.

Using a workflow system for event detection ensures that event detection is integrated into an application business process. The workflow system can also detect application events that are generated automatically without user involvement.

Figure 52 shows a workflow-based event detection mechanism. When a particular operation occurs, the workflow process is started. The event detection mechanism receives the information about the event and writes a record to the event store. The workflow process continues with other tasks.

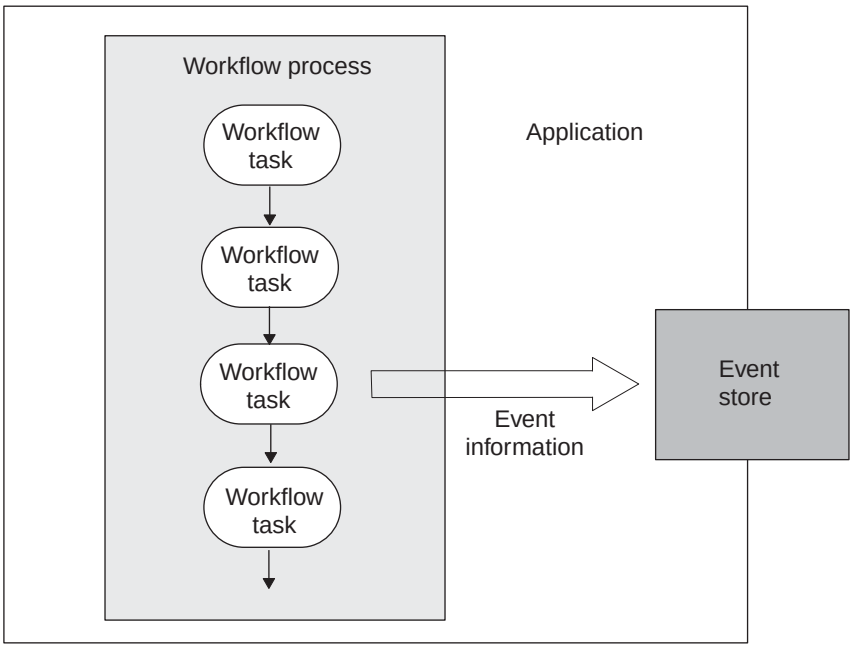

Figure 52. Workflow-based event detection

Database triggers

If the application has no built-in method for detecting events and the database that the application is running on provides database triggers, you may be able to implement row-level triggers to detect changes to application tables. The triggers are inserted in application tables that correspond to business object definitions supported by the connector.

With this mechanism, you also need to set up an event table in the application database to store the event records that the triggers generate. Whenever an application entity is created, updated, or deleted, a trigger inserts a row into the event table. Each row represents one event record, and the event table queues the events for processing by the connector.

Figure 53 shows a user action that updates an application Customer table. When the Customer table is updated, a trigger on the table executes and writes an event record to the event table in the application database.

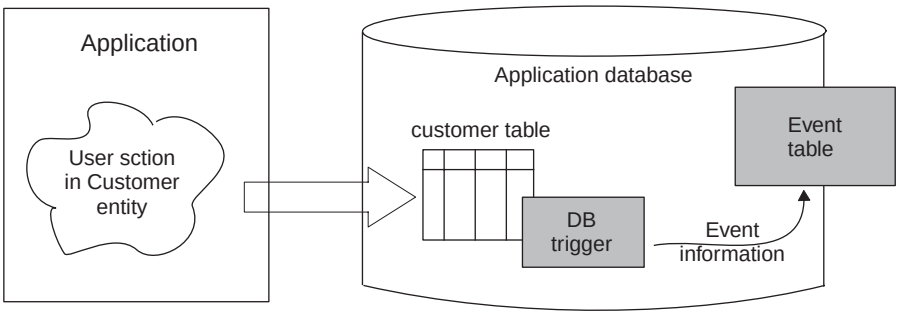

Figure 53. Event detection using database triggers

If you use database triggers, keep the following in mind:

- Make sure that any triggers you provide do not overwrite triggers already in use in the application.
- Make sure that the application is suitable for the use of triggers for event notification. For example, if an application has implemented complex business rules in its database, a simple trigger on a particular table might not accurately reflect the complete application event.
- A drawback to database triggers is that if table schemas change in the application database, you may need to modify the triggers that you have created. If table schemas change frequently and you have set up many database triggers, you may need to spend considerable time maintaining the triggers.

Event detection: standard behavior

An application event detection mechanism should take the following steps:

- Detect an event on an application entity for a business object supported by the connector.
- Create an event record. To create the record, the event detection mechanism should:
  - Set the name of the object to the complete name of the business object in the repository.
  - Set the verb to the action that occurred in the database.
  - Set the object key to the primary key of the application entity.
  - Generate a unique event identifier (ID).
  - Set the event priority.
  - Set the event timestamp.
  - Set the event status to Ready-for-Poll.
- Insert the completed event record into the event store.

  Note: An event detection mechanism can optionally query the event store for existing duplicate events before inserting a record for a new event. For more information, see "Filtering the event store for duplicate event records" on page 125.

Once event records are in the event store, the event store queues events for pickup by the connector's poll method. The event store should be internal to the application. If the application terminates unexpectedly, the event store can be restored to its preceding state when the application is restored, and the connector application-specific code can then pick up queued events.

The event detection mechanism should ensure data integrity between an application event and the event record written to the event store. For example, generation of an event record should *not* take place until *all* required data transactions for the event have completed successfully.

Subsequent sections provide the following information about issues to handle in the event detection mechanism:

- "Filtering the event store for duplicate event records"
- "Future event processing" on page 126

Filtering the event store for duplicate event records

The event detection mechanism can be implemented so that duplicate events are *not* saved in the event store. This behavior can minimize the amount of processing that the integration broker has to perform. As an example, if an application

updates a particular `Address` object several times between connector polls, all the events might be stored in the event store, and the connector will then create business objects for all events and send them to InterChange Server. To prevent this, the event detection mechanism can filter the events such that only a single Update event is stored.

Before storing a new event as a record in the event store, the event detection mechanism can query the event store for existing events that match the new event. The event detection mechanism should *not* generate a record for a new event in these cases:

Case 1	The business object name, verb, key, status, and `ConnectorId` (if applicable) in a new event match those of another unprocessed event in the event store.
Case 2	The business object name, key, and status for a new event match an unprocessed event in the event table; in addition, the verb for the new event is Update, and the verb for the unprocessed event is Create.
Case 3	The business object name, key, and status for a new event match an unprocessed event in the event table; in addition, the verb in the unprocessed event in the event table is Create, and the verb in the new event is Delete. In this case, remove the Create record from the event store.

Note: If event detection is implemented with stored procedures and triggers, the stored procedures can perform the query before inserting records for new events.

Future event processing

The event detection mechanism can be set up to specify a date and time in the future to process an event. To implement this feature, you may need to set up an additional event store for these events. Event records in the future event store should include a date that identifies when they will be processed.

This feature is required for applications with records that include effective dates. As an example, suppose that an existing employee will receive a promotion in a month and that, at that time, he will receive a raise. Because the paperwork for his increased compensation is completed prior to the date of his promotion, the change to his status generates an event with an effective date, which is stored in the future event table.

Implementing event retrieval

For most connectors, the application-specific component of the connector implements the event retrieval mechanism. The connector developer does this as part of the connector design and implementation. This mechanism works in conjunction with the event detection mechanism, which detects entity changes and writes event records to the event store. Event retrieval transfers information about application events from the event store to the connector's application-specific component.

This section provides the following information about event retrieval:

Event retrieval mechanisms

Two common mechanisms use to retrieve event records from an event store are:

- Event callback mechanism—connectors can be notified of application events through an event-callback mechanism; however, few applications currently provide event callback APIs for application events.
- Polling mechanism—the most common type of event retrieval mechanism is a polling mechanism.

Using a polling mechanism

In a polling mechanism, the application provides a persistent event store, such as an database table or inbox, where it writes event records when changes to application entities occur. The connector periodically checks, or polls, the event store for changes to entities that correspond to business object definitions that the connector supports. In general, the only information about the business object that is kept in the event store is the type of operation and the key values of the application entity. As the connector processes the event, it retrieves the remainder of the application entity data. After the connector has processed the event, it removes the event record from the event store and places it in an archive store.

To implement a polling mechanism to perform event retrieval, the connector's application-specific component uses a *poll method*, called the `pollForEvents()` method. The poll method checks the event store, retrieves new events, and processes each event before returning.

This section provides the following information about the poll method:

- "Polling interval"
- "Event polling: standard behavior"

Polling interval

The connector framework calls the poll method at a specified *polling interval* as defined by the `PollFrequency` connector configuration property. This property is initialized at connector installation time with Connector Configurator. Typically, the polling interval is about 10 seconds.

Note: If your connector does not need to poll to retrieve event information, polling can be turned off by setting the `PollFrequency` property to zero (0).

Therefore, the connector framework calls the `pollForEvents`() method in either of the following conditions:

- The `PollFrequency` is set to a value greater than zero.
- The connector startup script specifies a value for the `-fPollFreq` option.

Event polling: standard behavior

Figure 54 illustrates the basic behavior of a poll method:

1. The connector framework calls the application-specific component's `pollForEvents`() method to begin polling.
2. The `pollForEvents`() method checks the event store in the application for new events and retrieves the events.
3. The poll method then queries the connector framework to determine whether an event has subscribers.
4. If an event has subscribers, the poll method retrieves the complete set of data for the business object from the application.
5. The poll method sends the business object to the connector framework, which routes it to its destination (such as InterChange Server).

Each time the poll method is called, it checks for and retrieves new events, determines whether the event has subscribers, retrieves application data for events with subscribers, and sends business objects to InterChange Server.

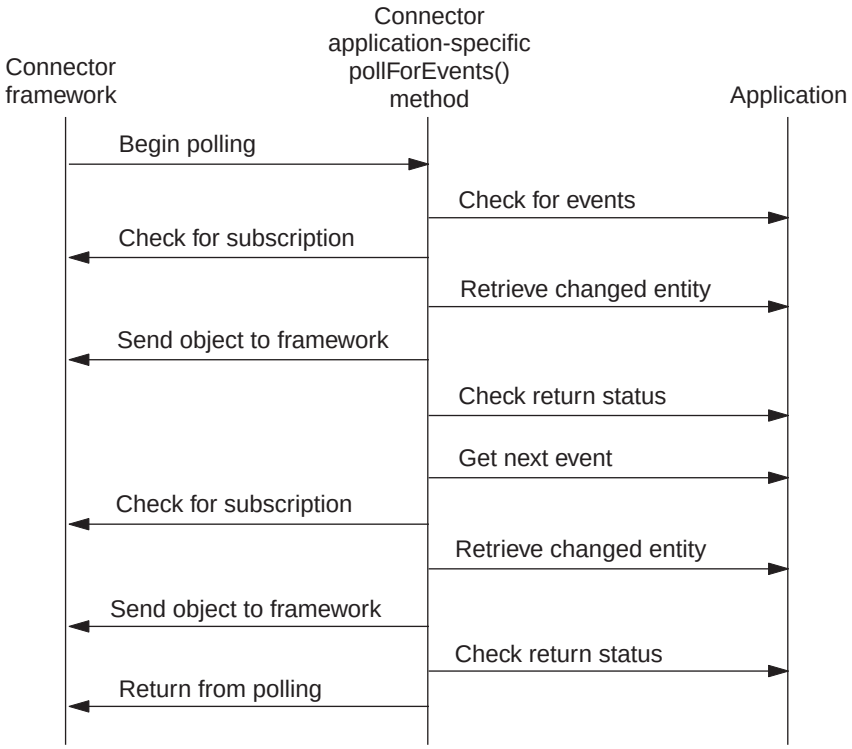

Figure 54. Basic behavior of pollForEvents() method

For information on how to implement the `pollForEvents()` method, see "Implementing the poll method" on page 128.

Implementing the poll method

Regardless of whether the application provides is an event store in a table, inbox, or other location, the connector must poll periodically to retrieve event information. The connector's poll method, `pollForEvents()`, polls the event store, retrieves event records, and processes events. To process an event, the poll method determines whether the event has subscribers, creates a new business object containing application data that encapsulates the event, and sends the business object to the connector framework.

Note: If your connector will be implementing request processing but *not* event notification, you might not need to fully implement `pollForEvents()`. However, since the poll method is defined with a default implementation in the Java connector library, polling is already implemented. If you want to disable polling, you can implement a stub for this method.

This section provides the following information on how to implement the `pollForEvents()` method:

- "Basic logic for pollForEvents()" on page 129
- "Other polling issues" on page 129

Basic logic for pollForEvents()

The `pollForEvents()` method typically uses a basic logic for event processing. Figure 55 shows a flow chart of the poll method's basic logic.

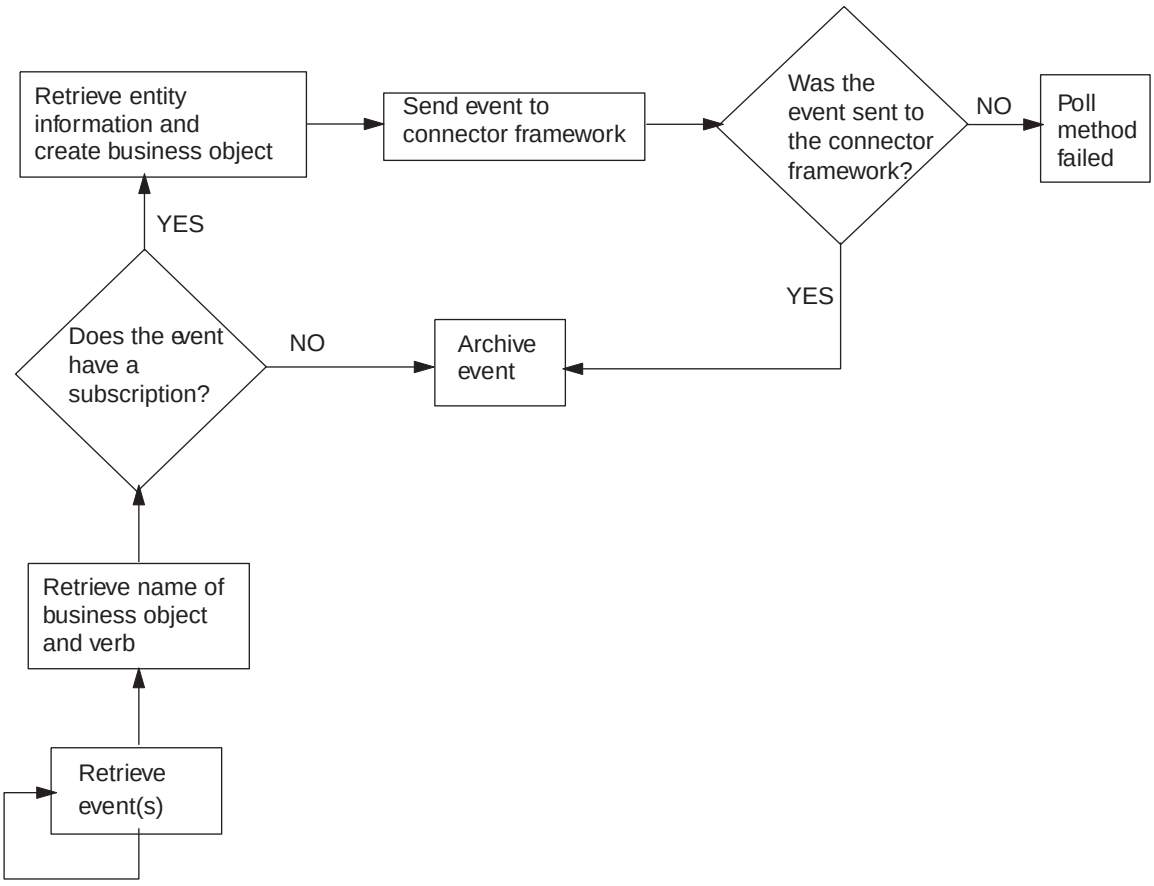

Figure 55. Flow chart for basic logic of pollForEvents()

For an implementation of this basic polling logic, see "Implementing an event-notification mechanism" on page 176."Implementing an event-notification mechanism" on page 189.

Note: For the event-status values that occur in the flow of the poll method, see Table 129 on page 315.Table 137 on page 333

Other polling issues

This section provides information on the following polling issues:

Archiving events

Once a connector has processed an event, it can archive the event. Archiving processed or unsubscribed events ensures that events are not lost. Archiving usually involves the following steps:

- Copy the event record from the event store to the archive store.

The *archive store* serves the same basic purpose as an event store: it saves archive records in a persistent cache until the connector can process them. An *archive record* contains the same basic information as an event record.

- Update the event status of the event in the archive store.

  The archive record should be updated with one of the event-status values in Table 49..
- Delete the event record from the event store.

Table 49. Event-status values in an archive record

Status	Description
Success	The event was detected, and the connector created a business object for the event and sent the business object to the connector framework.
Unsubscribed	The event was detected, but there were no subscriptions for the event, so the event was not sent to the connector framework and on to the integration broker.
Error	The event was detected, but the connector encountered an error when trying to process the event. The error occurred either in the process of building a business object for the event or in sending the business object to connector framework.

This section provides the following information about event archiving:

- "Creating an archive store"
- "Configuring a connector for archiving"
- "Accessing the archive store"

Creating an archive store: If the application provides archiving services, you can use those; otherwise, an archive store is usually implemented using the same mechanism as the event store:

- For an event-notification mechanism that uses database triggers, one way to set up event archiving is to install a delete trigger on the event table. When the connector's application-specific component deletes a processed or unsubscribed event from the event table, the delete trigger moves the event to the archive table. For information on event tables, see "Event table" on page 119.

  Note: If a connector uses an event table, an administrator might need to clean up the archive periodically.
- With an email event notification scheme, archiving might consist of moving a message to a different folder. A folder called `Archive` might be used for archiving event messages.

Configuring a connector for archiving: Archiving can have performance impact in the form of the archive store and moving the event records into this store. Therefore, you might want to design event archiving to be configurable at install time, so that a system administrator can control whether events are archived. To make archiving configurable, you can create a connector-specific configuration property that specifies whether the connector archives unsubscribed events. IBM suggests a name of `ArchiveProcessed` for this configuration property. If the configuration property specifies no archiving, the connector application-specific component can delete or ignore the event. If the connector is performance-constrained or the event volume is extremely high, archiving events is not required.

Accessing the archive store: A connector performs archiving as part of the event processing in its poll method, `pollForEvents()`. Once a connector has processed an

event, the connector must move the event to an archive store whether or not the event was successfully delivered to the connector framework. Events that have no subscriptions are also moved to the archive. Archiving processed or unsubscribed events ensures that events are not lost.

Your poll method should consider archiving an event when any of the following conditions occur:

- When the poll method has processed the event and the connector framework has delivered the business object
- When no subscriptions exist for the event

Note: If a connector uses an event table, an administrator might need to clean up the archive periodically. For example, the administrator may need to truncate the archive to free disk space.

Threading issues

Java connectors must be thread safe. The connector framework can use multiple threads to perform event delivery (execution of the `pollForEvents()` method) and request processing (execution of the `doVerbFor()` method).

Processing events by event priority

Event priority enables the connector poll method to handle situations where the number of events in the event store exceeds the maximum number of events the connector retrieves in a single poll. In this type of polling implementation, the poll method polls and processes events in order of priority. Event priority is defined as an integer value in the range 0 - n, with 0 as the highest priority.

To process events by event priority, the following tasks must be implemented in the event notification mechanism:

- The event detection mechanism must assign a priority value to an event record when it saves it to the event store.
- The event retrieval mechanism (the polling mechanism) must specify the order in which it retrieves event records to process, based on the event priority.

Note: As events are picked up, event priority values are *not* decremented. In rare circumstances, this might lead to low priority events being not picked up.

The following example SQL SELECT statement shows how a connector might select event records based on event priority. The SELECT statement sorts the events by priority, and the connector processes each event in turn.

```
SELECT event_id, object_name, object_verb, object_key
FROM event_table
WHERE event_status = 0 ORDER BY event_priority
```

The logic for a poll method is then the same as discussed in "Basic logic for pollForEvents()" on page 129.

Event distribution

The event detection and retrieval mechanisms can be implemented so that multiple connectors can poll the same event store. Each connector can be configured to process certain events, create specific business objects and pass those business objects to InterChange Server. This can streamline the processing of certain types of events and increase the transfer of data out of an application.

To implement event distribution so that multiple connectors can poll the event store, do the following:

- Add a column to the event record for an integer connector identifier (ID), and design the event detection mechanism to specify which connector will pick up the event.

  This might be done per application entity. For example, the event detection mechanism might specify that all `Customer` events be picked up by the connector that has the `connectorId` field set to 4.
- Add an application-specific connector property named `ConnectorId`. Assign each connector a unique identifier and store this value in its `ConnectorId` property.
- Implement the poll method to query for the value of the `ConnectorId` property. If the property is not set, the poll method can retrieve *all* event records from the event store as usual. If the property is set to a connector identifier value, the poll method retrieves *only* those events that match the `ConnectorId`.

Special considerations for event processing

This section contains the following information about event processing:

- "Processing Delete events"
- "Using guaranteed event delivery" on page 133

Processing Delete events

An application can support one of the following types of delete operations:

- Physical delete—Data is physically deleted from the database.
- Logical delete—A status column in a database entity is set to an inactive or invalid status, but the data is not deleted from the database.

It may be tempting to implement delete event processing in a manner that is consistent with the application. For example, when an application entity is deleted, a connector poll method for an application that supports physical deletes might publish a business object with the Delete verb. A connector poll method for an application that supports logical deletes might publish a business object with the Update verb and the status value changed to inactive.

Problems can arise with this approach when a source application and a destination application support different delete models. Suppose that the source application supports logical delete and the destination application supports physical delete. Assume that an enterprise is synchronizing between the source and destination applications. If the source connector sends a change in status (in other words, a delete event) as a business object with the Update verb, the destination connector might be unable to determine that the business object actually represents a delete event.

Therefore, event publishing must be designed so that source connectors for both types of applications can publish delete events in such a way that destination connectors can handle the events appropriately. The Delete verb in an event notification business object should represent an event where data was deleted, whether the delete operation was a physical or logical delete. This ensures that destination connectors will be correctly informed about a delete event.

This section provides the following information on how to implement event processing for delete events:

- "Setting the verb in the event record" on page 133

- "Setting the verb in the business object"
- "Setting the verb during mapping"

Setting the verb in the event record

The event detection mechanism for both logical and physical delete connectors should set the verb in the event record to Delete:

- For a physical delete connector, this is the standard implementation.
- For a connector whose application supports logical deletes, the event detection mechanism must be designed to determine when update events actually represent deletion of data.

  In other words, it must differentiate update events for modified entities from update events for logically deleted entities. For logically deleted entities, the event detection mechanism should set the verb in the event record to Delete even if the event in the application was an Update event that updated a status column.

Setting the verb in the business object

The poll method for both logical and physical delete connectors should generate a business object with the Delete verb:

- If the application supports logical deletes, the connector poll method retrieves the delete event from the event store, creates an empty business object, sets the key, sets the verb to Delete, and sends the business object to the connector framework.

  For hierarchical business objects, the connector should *not* send deleted children. The connector can constrain queries to not include entities with status of inactive, or child business objects with a status of inactive can be removed in mapping.
- If the application supports physical deletes, the connector might not be able to retrieve the application data. In this case, the connector poll method retrieves the delete event from the event store, creates an empty business object, sets the key values, sets the values of other attributes to the special Ignore value (`CxIgnore`), sets the verb in the business object to Delete, and sends the business object to the connector framework.

Setting the verb during mapping

WebSphere InterChange Server

Mapping between the application-specific business object and the generic business object should map the verb as Delete. This ensures that the correct information about an event is sent to the collaboration, which may perform special processing based on the verb.

Follow these recommendations for relationship tables:

- For delete events for a logical delete application, leave relationship entries in the relationship table.
- For delete events for a physical delete application, delete relationship entries from the relationship table.

Using guaranteed event delivery

The guaranteed-event-delivery feature enables the connector framework to guarantee that events are never sent twice between the connector's event store and the integration broker.

Important: This feature is available *only* for JMS-enabled connectors; that is, those connectors that use Java Messaging Service (JMS) to handle queues for their message transport. A JMS-enabled connector always has its `DeliveryTransport` connector property set to `JMS`. When the connector starts, it uses the JMS transport; all subsequent communication between the connector and the integration broker occurs through this transport. The JMS transport ensures that the messages are eventually delivered to their destination.

Without use of the guaranteed-event-delivery feature, a small window of possible failure exists between the time that the connector publishes an event (when the connector calls the `gotApplEvent()` method within its `pollForEvents()` method) and the time it updates the event store by deleting the event record (or perhaps updating it with an "event posted" status). If a failure occurs in this window, the event has been sent but its event record remains in the event store with a "ready for poll" status. When the connector restarts, it finds this event record still in the event store and sends it, resulting in the event being sent twice.

You can provide the guaranteed-event-delivery feature to a JMS-enabled connector in one of the following ways:

- With the *container-managed-events* feature: If the connector uses a JMS event store (implemented as a JMS source queue), the connector framework act as a container and manage the JMS event store. For more information, see "Guaranteed event delivery for connectors with JMS event stores."
- With the *duplicate-event-elimination* feature: The connector framework can use a JMS monitor queue to ensure that no duplicate events occur. This feature is usually used for a connector that uses a non-JMS event store (for example, implemented as a JDBC table, Email mailbox, or flat files). For more information, see "Guaranteed event delivery for connectors with non-JMS event stores" on page 136.

Guaranteed event delivery for connectors with JMS event stores

If the JMS-enabled connector uses JMS queues to implement its event store, the connector framework can act as a ″container″ and manage the JMS event store (the JMS source queue). One of the roles of JMS is to ensure that once a transactional queue session starts, the messages are cached there until a commit is issued; if a failure occurs or a rollback is issued, the messages are discarded. Therefore, in a single JMS transaction, the connector framework can remove a message from a source queue and place it on the destination queue. This *container-managed-events* feature of guaranteed event delivery enables the connector framework to guarantee that events are never sent twice between the JMS event store and the destination's JMS queue.

This section provides the following information about use of the guaranteed-event-delivery feature for a JMS-enabled connector that has a JMS event store:

- "Enabling the feature for connectors with JMS event stores"
- "Effect on event polling" on page 136

Enabling the feature for connectors with JMS event stores: To enable the guaranteed-event-delivery feature for a JMS-enabled connector that has a JMS event store, set the connector configuration properties shown in Table 50..

Table 50. Guaranteed-event-delivery connector properties for a connector with a JMS event store

Connector property	Value
`DeliveryTransport`	`JMS`
`ContainerManagedEvents`	`JMS`
`PollQuantity`	The number of events to processing in a single poll of the event store
`SourceQueue`	Name of the JMS source queue (event store) which the connector framework polls and from which it retrieves events for processing **Note:** The source queue and other JMS queues should be part of the same queue manager. If the connector's application generates events that are stored in a different queue manager, you must define a remote queue definition on the remote queue manager. WebSphere MQ can then transfer the events from the remote queue to the queue manager that the JMS-enabled connector uses for transmission to the integration broker. For information on how to configure a remote queue definition, see your IBM WebSphere MQ documentation.

Note: A connector can use *only one* of these guaranteed-event-delivery features: container managed events or duplicate event elimination.Therefore, you cannot set the `ContainerManagedEvents` property to `JMS` *and* the `DuplicateEventElimination` property to `true`.

In addition to configuring the connector, you must also configure the data handler that converts between the event in the JMS store and a business object. This data-handler information consists of the connector configuration properties that Table 51 summarizes.

Table 51. Data-handler properties for guaranteed event delivery

Data-handler property	Value	Required?
`MimeType`	The MIME type that the data handler handles. This MIME type identifies which data handler to call.	Yes
`DHClass`	The full name of the Java class that implements the data handler	Yes
`DataHandlerConfigMOName`	The name of the top-level meta-object that associates MIME types and their data handlers	Optional

Note: The data-handler configuration properties reside in the connector configuration file with the other connector configuration properties.

End users that configure a connector that has a JMS event store to use guaranteed event delivery must be instructed to set the connector properties as described in Table 50 *and* Table 51.. To set these connector configuration properties, use the Connector Configurator tool. Connector Configurator displays the connector

properties in Table 50 on its Standard Properties tab. It displays the connector properties in Table 51 on its Data Handler tab.

Note: Connector Configurator activates the fields on its Data Handler tab only when the `DeliveryTransport` connector configuration property is set to `JMS` and `ContainerManagedEvents` is set to `JMS`.

For information on Connector Configurator, see Appendix B, "Connector Configurator," on page 495.Appendix B, "Connector Configurator," on page 527.

Effect on event polling: If a connector uses guaranteed event delivery by setting `ContainedManagedEvents` to `JMS`, it behaves slightly differently from a connector that does not use this feature. To provide container-managed events, the connector framework takes the following steps to poll the event store:

1. Start a JMS transaction.
2. Read a JMS message from the event store.

   The event store is implemented as a JMS source queue. The JMS message contains an event record. The name of the JMS source queue is obtained from the `SourceQueue` connector configuration property.
3. Call the appropriate data handler to convert the event to a business object.

   The connector framework calls the data handler that has been configured with the properties in Table 51 on page 135..
4. When a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Business Integration Message Broker) or WebSphere Application Server is the integration broker, convert the business object to a message based on the configured wire format (XML).
5. Send the resulting message to the JMS destination queue.

   WebSphere InterChange Server
   The message sent to the JMS destination queue is the business object.

   Other integration brokers
   The message sent to the JMS destination queue is an XML message.
6. Commit the JMS transaction.

   When the JMS transaction commits, the message is written to the JMS destination queue and removed from the JMS source queue in the same transaction.
7. Repeat step 1 through 6 in a loop. The `PollQuantity` connector property determines the number of repetitions in this loop.

Important: A connector that sets the `ContainerManagedEvents` property is set to `JMS` does *not* call the `pollForEvents()` method to perform event polling. If the connector's base class includes a `pollForEvents()` method, this method is *not* invoked.

Guaranteed event delivery for connectors with non-JMS event stores

The connector framework can use *duplicate event elimination* to ensure that duplicate events do not occur. This feature is usually enabled for JMS-enabled connectors that use a non-JMS solution to implement an event store (such as a JDBC event

table, Email mailbox, or flat files). This duplicate-event-elimination feature of guaranteed event delivery enables the connector framework to guarantee that events are never sent twice between the event store and the destination's JMS queue.

Note: JMS-enabled connectors that use a JMS event store usually use the container-managed-events feature. However, they can use duplicate event elimination instead of container managed events.

This section provides the following information about use of the guaranteed-event-delivery feature with a JMS-enabled connector that has a non-JMS event store:

- "Enabling the feature for connectors with non-JMS event stores"
- "Effect on event polling" on page 136

Enabling the feature for connectors with non-JMS event stores: To enable the guaranteed-event-delivery feature for a JMS-enabled connector that has a non-JMS event store, you must set the connector configuration properties shown in Table 52..

Table 52. Guaranteed-event-delivery connector properties for a connector with a non-JMS event store

Connector property	Value
`DeliveryTransport`	`JMS`
`DuplicateEventElimination`	`true`
`MonitorQueue`	Name of the JMS monitor queue, in which the connector framework stores the ObjectEventId of processed business objects

Note: A connector can use *only one* of these guaranteed-event-delivery features: container managed events or duplicate event elimination.Therefore, you cannot set the `DuplicateEventElimination` property to `true` *and* the `ContainerManagedEvents` property to `JMS`.

End users that configure a connector to use guaranteed event delivery must be instructed to set the connector properties as described in Table 52.. To set these connector configuration properties, use the Connector Configurator tool. It displays these connector properties on its Standard Properties tab. For information on Connector Configurator, see Appendix B, "Connector Configurator," on page 495.Appendix B, "Connector Configurator," on page 527.

Effect on event polling: If a connector uses guaranteed event delivery by setting `DuplicateEventElimination` to `true`, it behaves slightly differently from a connector that does not use this feature. To provide the duplicate event elimination, the connector framework uses a JMS monitor queue to track a business object. The name of the JMS monitor queue is obtained from the `MonitorQueue` connector configuration property.

After the connector framework receives the business object from the application-specific component (through a call to `gotApplEvent()` in the `pollForEvents()` method), it must determine if the current business object (received from `gotApplEvents()`) represents a duplicate event. To make this

determination, the connector framework retrieves the business object from the JMS monitor queue and compares its ObjectEventId with the ObjectEventId of the current business object:

- If these two ObjectEventIds are the same, the current business object represents a duplicate event. In this case, the connector framework ignores the event that the current business object represents; it does *not* send this event to the integration broker.
- If these ObjectEventIds are *not* the same, the business object does *not* represent a duplicate event. In this case, the connector framework copies the current business object to the JMS monitor queue and then delivers it to the JMS delivery queue, all as part of the same JMS transaction. The name of the JMS delivery queue is obtained from the `DeliveryQueue` connector configuration property. Control returns to the connector's `pollForEvents()` method, after the call to the `gotApplEvent()` method.

For a JMS-enabled connector to support duplicate event elimination, you must make sure that the connector's `pollForEvents()` method includes the following steps:

- When you create a business object from an event record retrieved from the non-JMS event store, save the event record's unique event identifier as the business object's ObjectEventId attribute.

  The application generates this event identifier to uniquely identify the event record in the event store. If the connector goes down after the event has been sent to the integration broker but before this event record's status can be changed, this event record remains in the event store with an In-Progress status. When the connector comes back up, it should recover any In-Progress events. When the connector resumes polling, it generates a business object for the event record that still remains in the event store. However, because both the business object that was already sent and the new one have the same event record as their ObjectEventIds, the connector framework can recognize the new business object as a duplicate and not send it to the integration broker.

  A Java connector can use the `setDEEId()` method of the `CWConnectorBusObj` class to assign the event identifier to the ObjectEventId attribute, as follows:

  ```
  busObj.setDEEId(event_id);
  ```

- During connector recovery, make sure that you process In-Progress events *before* the connector begins polling for new events.

  Unless the connector changes any In-Progress events to Ready-for-Poll status when it starts up, the polling method does not pick up the event record for reprocessing.

Chapter 6. Message logging

This chapter presents information on message logging. A *message* is a string of information that the connector can send to an external connect log, where it can be reviewed by the system administrator or the developer to provide information about the runtime state of the connector. There are two different categories of messages that a connector can send to the connector log:

- Error or informational messages
- Trace messages

Messages can be generated within the connector code or obtained from a message file. This chapter contains the following sections:

Error and informational messages

A connector can send information about its state to a log destination. The following types of information are recommended for logging:

- Errors and fatal errors from your code to a log file.
- Warnings require a system administrator's attention, from your code to a log file.
- Informational messages such as:
  - Connector startup and termination messages
  - Important messages from the application

Although a connector can send informational or error messages, this logging process is referred to as *error logging*.

Note: These messages are independent of any trace messages defined for the connector.

Indicating a log destination

A connector sends its log messages into its log destination. The *log* is an external destination that is available for viewing by those needing to review the execution state of the connector. The log destination is defined at connector configuration time by the setting of the Logging field in the Trace/Log Files tab of Connector Configurator as one of the following:

- To File: The absolute pathname of an external file, which must reside on the same machine as the connector's process (with its connector framework and application-specific component)
- To console (STDOUT): The command prompt window generated when the connector startup script starts the connector

By default, the connector's log destination is set to the console, which indicates use of the startup script's command prompt window as the log destination. Set this log destination as appropriate for your connector.

WebSphere InterChange Server

You can also set the LogAtInterchangeEnd connector configuration property to indicate whether messages are also logged to the InterChange Server's log destination:

- Messages logged locally *only*: LogAtInterchangeEnd is false.
- Messages are logged both locally *and* sent to InterChange Server's log destination: LogAtInterchangeEnd is true.

By default, LogAtInterchangeEnd is set to false, so that messages are only logged locally. If messages are sent to InterChange Server, they are written to the destination specified for InterChange Server messages.

Note: Logging to InterChange Server's log destination also turns on email notification, which generates email messages for the MESSAGE_RECIPIENT parameter specified in the InterchangeSystem.cfg file when errors or fatal errors occur. As an example, when a connector loses its connection to its application, if LogAtInterchangeEnd is set to true, an email message is sent to the specified message recipient.

These connector properties are set with Connector Configurator. For more information on InterChange Server's message logging, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Sending a message to the log destination

Table 53 shows the ways that a connector sends an error, warning, and information message to its log destination.

Table 53. Methods for sending a message to the log destination

Connector library method	Description
logMsg() and generateMsg()	Takes as input a text string or a string generated from a message in a message file. Optionally, it can take a message-type constant to indicate whether the message is an error, warning, or informational. To generate a character string from the message text in a message file, use the generateMsg() method.
generateAndLogMsg()	Combines the functionality of the logMsg() and generateMsg() methods into a single call.

For more information on how to generate a message, see "Generating a message string" on page 145.

In the Java connector library, the logMsg(), generateMsg(), and generateAndLogMsg() methods are defined in the CWConnectorUtil class.

Both the generateMsg() and generateAndLogMsg() methods require a message type as an argument. This argument indicates the severity of the message. For more information, see "Generating a message string" on page 145.

Trace messages

Tracing is an optional troubleshooting and debugging feature that can be turned on for connectors. When tracing is turned on, system administrators can follow events as they work their way through the IBM WebSphere business integration system.

WebSphere InterChange Server

When InterChange Server is the integration broker, you can also use tracing on connector controllers, and other components of the InterChange Server system.

Tracing in an application-specific component allows you and other users of your connector code to monitor the behavior of the connector. Tracing can also track when specific connector functions are called by the connector framework. Trace messages that you provide for the connector application-specific code augment the trace messages provided for the connector framework.

Enabling tracing

By default, tracing on a connector is turned off. Tracing is turned on for a connector when the connector configuration property `TraceLevel` is set to a non-zero value in Connector Configurator. You can set `TraceLevel` to a value from 1 to 5 to obtain the appropriate level of detail. Level 5 tracing logs the trace messages of *all* lower trace levels.

WebSphere InterChange Server

Tip: For information on turning on tracing for connector controllers or for other components of the InterChange Server system, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Identifying a trace destination

A connector sends its trace messages into its trace destination, which is an external destination that is available for viewing by those needing to review the execution state of the connector. The trace destination is defined at connector configuration time by the setting of the Tracing field in the Trace/Log Files tab of Connector Configurator as one of the following:

- To File: The absolute pathname of an external file, which must reside on the same machine as the connector's process (with its connector framework and application-specific component)
- To console (STDOUT): The command prompt window generated when the connector startup script starts the connector

By default, the connector's trace destination is set to the console, which indicates use of the startup script's command prompt window as the trace destination. Set this trace destination as appropriate for your connector.

Sending a trace message to the trace destination

Table 54 shows the ways that a connector sends a trace message to its trace destination.

Table 54. Methods for sending a trace message to the trace destination

Connector library method	Description
`traceWrite()` and `generateMsg()`	Takes as input a text string or a string generated from a message in a message file and a trace-level constant to indicate the trace level. This method writes a trace message for the specified trace level or greater to the trace destination. To generate a character string from the message text in a message file, use the `generateMsg()` method with the message type set to `XRD_TRACE`.
`generateAndTraceMsg()`	Combines the functionality of the `traceWrite()` and `generateMsg()` methods into a single call.

For information on the `generateMsg()` method, see "Generating a message string" on page 145.

Note: It is *not* required that trace messages be localized in the message file. Whether trace messages are contained in a message file is left at the discretion of the developer. For more information, see "Locale-sensitive design principles" on page 56..

In the Java connector library, the `traceWrite()`, `generateMsg()`, and `generateAndTraceMsg()` methods are defined in the `CWConnectorUtil` class.

The `traceWrite()` and `generateAndTraceMsg()` require a trace level as an argument. This argument specifies the trace level to use for a trace message. When you turn on tracing at runtime, you specify a trace level at which to run the tracing. All trace messages in your code with trace levels at or below the runtime trace level are sent to the trace destination. For more information, see "Recommended content for trace messages" on page 142.

To specify a trace level to associate with a trace message, use a trace-level constant of the form `TRACELEVEL`*n* where *n* can be a trace level from 1 to 5. Trace-level constants are defined in the `CWConnectorLogAndTrace` class.

The `generateMsg()` method requires a message type as an argument. This argument indicates the severity of the message. Because trace messages do not have severity levels, you just use the `XRD_TRACE` message-type constant. Message-type constants are defined in the `CWConnectorLogAndTrace` class.

Note: The `generateAndTrace()` method does *not* require a message type as an argument. The method automatically assumes the `XRD_TRACE` message-type constant.

Recommended content for trace messages

You are responsible for defining what kind of information your connector returns at each trace level. Table 55 shows the recommended content for application-specific connector trace messages.

Table 55. Content of application-specific connector trace messages

Level	Content
0	Trace message that identifies the connector version. No other tracing is done at this level.

Table 55. Content of application-specific connector trace messages (continued)

Level	Content
1	Trace messages that: • Log status messages and identifying (key) information for each business object processed. • Record each time the `pollForEvents()` method is executed.
2	Trace messages that: • Identify the business object handlers used for each object the connector processes. • Log each time a business object is posted to InterChange Server, either from `gotApplEvent()` or `executeCollaboration()`. • Indicate each time a request business object is received.
3	Trace messages that: • Identify the foreign keys being processed (if applicable). These messages appear when the connector has encountered a foreign key in a business object or when the connector sets a foreign key in a business object. • Relate to business object processing. Examples of this include finding a match between business objects, or finding a business object in an array of child business objects.
4	Trace message that: • Identify application-specific information. Examples of this information include the values returned by the methods that process the application-specific information fields in business objects. • Identify when the connector enters or exits a function. These messages help trace the process flow of the connector. • Record any thread-specific processing. For example, if the connector spawns multiple threads, a message should log the creation of each new thread.
5	Trace message that: • Indicate connector initialization. Examples of this message include the value of each connector configuration property that has been retrieved from InterChange Server. • Detail the status of each thread the connector spawns while it is running. • Represent statements executed in the application. The connector log file contains all statements executed in the target application and the value of any variables that are substituted (where applicable). • Record business object dumps. The connector should output a text representation of a business object before it begins processing (showing the object the connector receives from the integration broker) and after it has processed the object (showing the object the connector returns to the integration broker).

Note: The connector should deliver all the trace messages at the specified trace level and lower.

For information on the content and level of detail for connector framework trace messages, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Message file

You can provide message input to the connector error logging or tracing method be as text strings or as references to a *message file*. A message file is a text file containing message numbers and message text. The message text can contain positional parameters for passing runtime data out of your connector. You can provide a message file by creating a file and defining the messages that you need.

WebSphere InterChange Server

Important: Do *not* add your messages to the InterChange Server message file, `InterchangeSystem.txt`. Access *only* existing messages from this system message file.

This section provides the following information about a message file:

- "Message format"
- "Name and location of a message file"
- "Generating a message string" on page 145

Message format

Within a message file, messages have the following format:

message number message text[EXPL]*explanation text*

The *message number* is an integer that uniquely identifies the message. This message number must appear on one line. The *message text* can span multiple lines, with a carriage return terminating each line. The *explanation text* is a more detailed explanation of the condition that causes the message to occur.

As an example of message text, a connector can call the following message when it determines that its version differs from the version of the connector infrastructure.

```
20017
Connector Infrastructure version does not match.
```

Messages can contain parameters whose values are replaced at runtime by values from the program. The parameters are positional and are indicated in the message file by a number in braces. For example, the following message has two parameters to record an unsubscribed event.

```
20026
Warning: Unsubscribed event: Object Name:{1}, Verb: {2}.
```

For information on how to provide values to message parameters, see "Using parameter values" on page 147.

Note: For additional examples of messages, see the InterChange Server message file `InterchangeSystem.txt`.

Name and location of a message file

A connector can obtain its messages from one of two message files:

- A connector message file is named *Appname*`Connector.txt` and is stored in the following subdirectory of the product directory:

  ```
  connectors\messages
  ```

For example, the connector message file for the IBM WebSphere Business Integration Adapter for Clarify is named `ClarifyConnector.txt`.

- The InterChange Server message file is named `InterchangeSystem.txt` and is located in the product directory.

When you generate a message, you can specify which of these two message files to extract a message from with a message-file constant. All methods that generate messages (see Table 56 on page 145) provide a parameter to specify which message file to use. For more information, see "Specifying a message number" on page 146.

WebSphere InterChange Server

If a connector message file does *not* exist, the InterChange Server message file `InterchangeSystem.txt` (located in the product directory) is used as the message file.

The connector message file should contain all text strings that the application-specific component uses. These strings include those for logging as well as hardcoded strings.

Note: Connector standards suggest that trace messages are *not* included in a connector message file because end users do not normally view them.

For an internationalized connector, it is important that text strings are isolated into the connector message file. This message file can be translated and the messages can then be easily available in different languages. The name of the translated connector message file should include the name of the associated locale, as follows:

Appname`Connector_`*ll*`_`*TT*`.txt`

In the preceding line, *ll* is the two-letter abbreviation for the locale (by convention in lowercase letters) and *TT* is the two-letter abbreviation for the territory (by convention in uppercase letters). For example, the version of the connector message file for the WBI Adapter for Clarify that contains U.S. English messages would have the following name:

`ClarifyConnector_en_US.txt`

At runtime, the connector framework locates the appropriate message file for the connector framework locale from the `connectors\messages` subdirectory. For example, if the connector framework's locale is U.S. English (`en_US`), the connector framework retrieves messages from the *Appname*`Connector_en_US.txt` file.

For more information on how to internationalize the text strings of a connector, see "An internationalized connector" on page 55..

Generating a message string

The methods in Table 56 retrieve a predefined message from a message file, format the text, and return a generated message string.

Table 56. Methods that generate a message string

Message method	Description
`generateMsg()`	Generates a message of the specified severity from a message file. You can use the message as input to the `logMsg()` or `traceWrite()` method.

Table 56. Methods that generate a message string (continued)

Message method	Description
`generateAndLogMsg()`	Generates a message of the specified severity from a message file and sends it to the log destination
`generateAndTraceMsg()`	Generates a trace message from a message file and sends it to the log destination

Tip: Before using `generateMsg()` for tracing, check that tracing is enabled with the `isTraceEnabled()` method. If tracing is *not* enabled, you need not generate the trace message.

In the Java connector library, the `generateMsg()`, `generateAndLogMsg()`, and `generateAndTraceMsg()` methods are defined in the `CWConnectorUtil` class

The message-generation methods in Table 56 require the following information:

- "Specifying a message number"
- "Specifying a message type" on page 147
- "Using parameter values" on page 147 (optional)

Specifying a message number

The methods in Table 56 require a message number as an argument. This argument specifies the number of the message to obtain from the message file. As described in "Message format" on page 144, each message in a message file must have a unique integer message number (identifier) associated with it. The methods in Table 56 search the message file for the specified message number and extract the associated message text.

To indicate which message file these methods search in, you specify an integer message-file constant as an argument, as Table 57 shows.

Table 57. Message-file constants

Message-file constant	Description
`INFRASTRUCTURE_MESSAGE_FILE`	Search the InterChange Server message file (`InterchangeSystem.txt`) for the specified message number. **Note:** This message-file constant is valid *only* when the integration broker is InterChange Server.
`CONNECTOR_MESSAGE_FILE`	Search the connector message file for the specified message number.

In the Java connector library, the message-file constants are defined in the `CWConnectorLogAndTrace` class.

The IBM WebSphere business integration system generates the date and time and displays the following message:

```
[1999/05/28 11:54:15.990] [ConnectorAgent ConnectorName]
Error 1100: Failed to connect to application
```

Note: If the connector logs to its local log file, the connector infrastructure adds the timestamp.

WebSphere InterChange Server

If the connector logs to InterChange Server, InterChange Server adds the timestamp.

Specifying a message type

The methods in Table 56 also require a message type as an argument. This argument indicates the severity of the message. Table 58 lists the valid message types and their associated message-type constants.

Table 58. Message types

Message type	Severity level	Description
XRD_FATAL	Fatal Error	Indicates an error that stops program execution.
XRD_ERROR	Error	Indicates a error that should be investigated.
XRD_WARNING	Warning	Indicates a condition that might represent a problem but that can be ignored.
XRD_INFO	Informational	Information message only; no action required.
XRD_TRACE	--	Use for trace messages.

To specify a message type to associate with a message, use one of the message-type constants in Table 58 as an argument to the message-generation method, as follows:

- For a log message, use a message-type constant that indicates the message severity (in decreasing level of severity): XRD_FATAL, XRD_ERROR, XRD_WARNING, XRD_INFO.
- For a trace message, use the XRD_TRACE message-type constant.

In the Java connector library, the generateAndTraceMsg() method does *not* require a message type for trace messages. Although the Java connector library supports a deprecated version of generateAndTraceMsg() that requires the message type, the nondeprecated version of this method automatically specifies the XRD_TRACE message type; therefore, you do *not* need to provide it as an argument.

Message-type constants are defined in the CWConnectorLogAndTrace class.

Using parameter values

With the message-generation methods in Table 56, you can specify an optional number of values for message-text parameters. The number of parameter values must match the number of parameters defined in the message text. For information on how to define parameters in a message, see "Message format" on page 144.

To specify parameter values, you must include the following arguments:

- An argument count to indicate the number of parameters within the message text; to determine the number, refer to the message in the message file.
- A comma-separated list of parameter values; each parameter is represented as a character string.

Suppose you have the following informational message in your connector message file that contains one parameter:

```
2887
Initializing connector {1}
```

Because this message contains a single parameter, a call to one of the message-generation methods must specify an argument count of 1 and then provide the name of the connector as a character string. In the code fragment below, `generateAndLogMsg()` is called to format a message that contains one parameter and send this message to the log:

```
String val = CWConnectorUtil.getConfigProp("ConnectorName");
CWConnectorUtil.generateAndLogMsg(2887, CWConnectorLogAndTrace.XRD_INFO,
     CWConnectorUtil.CONNECTOR_MESSAGE_FILE, 1, val);
```

The parameter value of `val` is combined with the message in the message file. If `val` is set to `MyConnector`, the resulting message is:

```
[1999/05/28 11:54:15.990] [ConnectorAgent MyConnector]
       Info 2887: Initializing connector MyConnector
```

You can also locate trace messages in the connector message file.

Chapter 7. Implementing a Java connector

This chapter presents information on how to implement a connector's application-specific component in Java. It provides language-specific details for the general tasks discussed in earlier chapters of this guide.

This chapter contains the following sections:

Extending the Java connector base class

In the Java connector library, the connector base class is named `CWConnectorAgent`. The `CWConnectorAgent` class provides methods for startup, subscription checking, business object subscription delivery, and shut down. To implement your own connector, you extend this connector base class to create your own *connector class*.

Note: For general information about the methods of the connector base class, see "Extending the connector base class" on page 68..

To derive a connector class for a Java connector, follow these steps:

1. Create a connector class that extends the `CWConnectorAgent` class. Name this connector class:

   connectorName`Agent.java`

   where *connectorName* uniquely identifies the application or technology with which the connector communicates. For example, to create a connector for a Baan application, you create a connector class called `BaanAgent`.
2. In the connector-class file, define a package name to contain your connector. A connector package name has the following format:

   `com.crossworlds.connectors.`*connectorName*

   where *connectorName* is the same as defined in step 1 above. For example, the package name for the Baan connector would be defined in the connector-class file as follows:

   ```
   package com.crossworlds.connectors.Baan;
   ```
3. Ensure that the connector-class file imports the following classes:

   ```
   com.crossworlds.cwconnectorapi.*;
   com.crossworlds.cwconnectorapi.exceptions.*;
   ```

   If you create several files to hold the connector-class code, you must import these classes into *every* connector file.
4. Implement the appropriate base-class methods for the connector's application-specific component. For more information on how to create these base-class methods, see Table 59..

Table 59. Extending base-class methods of the CWConnectorAgent class

CWConnectorAgent method	Description	For more information
`agentInit()`	Initializes the application-specific component of the connector.	"Initializing the connector" on page 150
`getVersion()`	Obtain the version of the connector.	"Checking the connector version" on page 151
`getConnectorBOHandlerForBO()`	Obtain the business-object handler for the business objects.	"Obtaining the Java business object handler" on page 153
`getEventStore()`	Obtain the event-store object for the connector.	"CWConnectorEventStoreFactory interface" on page 178
`doVerbFor()`	Process the request business object by performing its verb operation.	"Creating a business object handler" on page 154
`pollForEvents()`	Poll event store to obtain application events and send them to the connector framework.	"Implementing an event-notification mechanism" on page 176
`terminate()`	Perform cleanup operations for the connector shut down.	"Shutting down the connector" on page 202

Beginning execution of the connector

When the connector is started, the connector framework instantiates the associated connector class and then calls the connector class methods in Table 60..

Table 60. Beginning execution of the connector

Initialization task		For more information
1.	Initialize the connector to perform any necessary initialization for the connector, such as opening a connection to the application.	"Initializing the connector" on page 150
2.	For each business object that the connector supports, obtain the business object handler.	"Obtaining the Java business object handler" on page 153

Initializing the connector

To begin connector initialization, the connector framework calls the initialization method, `agentInit()`, in the connector base class, `CWConnectorAgent`. This method performs initialization steps for the connector's application-specific component.

Important: As part of the implementation of your connector class, you *must* implement an `agentInit()` method for your connector.

As discussed in "Initializing the connector" on page 65,, the main tasks of the `agentInit()` initialization method include:

- "Establishing a connection" on page 151
- "Checking the connector version" on page 151
- "Recovering In-Progress events" on page 151

In addition to the above topics, this section provides an example Java `agentInit()` method in "Example Java initialization method" on page 152.

Important: During execution of the initialization method, business object definitions and the connector framework's subscription list are *not* yet available.

Establishing a connection

The main task of the `agentInit()` initialization method is to establish a connection to the application. It executes successfully if the connector succeeds in opening a connection. If the connector *cannot* open a connection, the initialization method must throw the `ConnectionFailureException` exception to indicate the cause of the failure. The connector might also need to log into the application. If this logon attempt fails, the initialization method must throw the `LogonFailedException` to indicate the cause of the failure. The steps in Table 64 on page 157 outline how to throw either of these initialization exceptions.

Note: For an overview of the steps in an initialization method, see "Establishing a connection" on page 65..

Checking the connector version

The `getVersion()` method returns the version of the connector's application-specific component.

Note: For a general description of `getVersion()`, see "Checking the connector version" on page 65..

In the Java connector library, the `getVersion()` method is defined in the `CWConnectorAgent` class. This class provides a default implementation of `getVersion()` that obtains the version from the Java manifest file. You can override `getVersion()` to provide a different implementation.

For example, the following code sample implements `getVersion()` to return a string indicating the version of the connector.

```
public String getVersion(){
 // get version from manifest file, or from string

  String version = "1.0.0";
  return version;
}
```

Recovering In-Progress events

The Java connector library provides the `CWConnectorEventStore` class to represent an event store. To recover In-Progress event records in the event store, the Java connector library provides the method in Table 61..

Table 61. Method for recovering In-Progress events

Java connector library class	Method
`CWConnectorEventStore`	recoverInProgressEvents()

The `recoverInProgressEvents()` method implements the recovery behavior for In-Progress events. However, the `CWConnectorEventStore` class does *not* provide a default implementation for this method. One possible recovery behavior is based on the `InDoubtEvents` connector configuration property and is outlined in Table 23 on page 66..

Note: For a general discussion of how to recover In-Progress events, see"Recovering In-Progress events" on page 65..

If the recovery process fails, the initialization method must throw the `InProgressEventRecoveryFailedException` to indicate the cause of the failure. The steps in Table 64 on page 157 outline how to throw this initialization exception.

Figure 56 shows a fragment of the agentInit() method that uses recoverInProgressEvents() to recover the In-Progress events.

```
// instantiate event-store factory
evtFac=new MyEventStoreFactoryInstance();

// instantiate event store
Object evto=evtFac.getEventStore();
CWConnectorEventStore evts=(CWConnectorEventStore)evto;

// check for any leftover In-Progress events
String inDoubtEvents=CWConnectorUtil.getConfigProp(
   "InDoubtEvents");

// In case the InDoubtEvents property is not set, use
// FailOnStartup as default.
if (inDoubtEvents == null || inDoubtEvents.equals(""))
   inDoubtEvents="FailOnStartup";

// recover In-Progress events
if (evts.recoverInProgressEvents() == FAIL
      || inDoubtEvents.equals("FailOnStartup") ) {

      // log a fatal error

      // throw an exception to terminate agentInit()
      throw new InProcessEventRecoveryFailureException()
   }
}
```

Figure 56. Recovering in-progress events

In Figure 56,, the MyEventStoreFactoryInstance class is an example of an extension of the CWConnectorEventStoreFactory class, whose getEventStore() method provides access to the event store.

Example Java initialization method

For a Java connector, the agentInit() method provides the initialization for the connector's application-specific component. This method does not return a value but throws special exceptions to indicate common initialization errors. Figure 57 shows a simple agentInit() method that obtains connector properties and establishes a connection to the application.

```
public agentInit()
   throws PropertyNotSetException, ConnectionFailureException,
   InProgressEventRecoveryFailedException, LogonFailedException
{

   CWConnectorUtil.traceWrite(CWConnectorLogAndTrace.LEVEL4,
      "Entering Connector agentInit()");

   int status = CWConnectorConstant.SUCCEED;
   connectorProperties =
      CWConnectorUtil.getAllConnectorAgentProperties();

   ExampleConnection userConnect = new Connection();

   // get Connector Configuration Properties to establish Connection
   String connectString =
      (String)connectorProperties.get("ConnectString");
   String userName =
      (String)connectorProperties.get("ApplicationUserName");
   String userPassword =
      (String)connectorProperties.get("ApplicationPassword");

   if(connectString == null || connectString.equals("")
         || userName==null || userPassword==null )
      {
      throw new PropertyNotSetException();
      }

   // Use Configuration Values to log into Application
   try
   {
      boolean loginSuccessful = userConnect.login(connectString,
         userName, userPassword);

      if(loginSuccessful)
         CWConnectorUtil.generateAndLogMsg(30000,CWConnectorLogAndTrace.XRD_INFO,);
   }
   catch(ExampleAppException se)
   {
      CWConnectorUtil.generateAndLogMsg(30001,
            CWConnectorLogAndTrace.XRD_ERROR,0,1,path);
   }

}
```

Figure 57. Initializing a Java connector

Note: For `agentInit()` code fragment that recovers In-Progress events, see Figure 56 on page 152..

Obtaining the Java business object handler

In a Java connector, the business-object-handler base class is `CWConnectorBOHandler`. To obtain an instance of a business object handler for a supported business object, the connector framework calls the `getConnectorBOHandlerForBO()` method, which is defined as part of the `CWConnectorAgent` class.

Note: For general information about the `getConnectorBOHandlerForBO()` method, see "Obtaining the business object handler" on page 66.. For a general discussion of how to design business object handlers, see "Designing business object handlers" on page 79..

The default implementation of `getConnectorBOHandlerForBO()` in the `CWConnectorAgent` class returns a business object handler for a business-object-handler base class named `ConnectorBOHandler`. If you name your extended business-object-handler base class `ConnectorBOHandler`, you do not need to override the `getConnectorBOHandlerForBO()` method. However, if you name your extended business-object-handler base class some other than `ConnectorBOHandler`, you must override `getConnectorBOHandlerForBO()` to return an instance of your extended business-object-handler base class.

The number of business object handlers that the connector framework obtains through its calls to the `getConnectorBOHandlerForBO()` method depends on the overall design for business object handling in your connector:

- If the connector is metadata-driven, it can be designed to use a generic, metadata-driven business object handler.

  Figure 58 contains an implementation of the `getConnectorBOHandlerForBO()` method that returns a metadata-driven business object handler. It calls the constructor for the `ExampleBOHandler` class, which instantiates a single business-object-handler base class that handles *all* the business objects supported by the connector.
- If some or all application-specific business objects require special processing, then you must set up multiple business object handlers for those objects.

Important: During execution of the `getConnectorBOHandlerForBO()` method, the business object class methods are not yet available.

Figure 58 calls the constructor for the `ExampleConnectorBOHandler` class to instantiate a single business-object-handler base class that handles *all* the business objects supported by the connector.

```
public CWConnectorBOHandler getConnectorBOHandlerForBO(String BOName){
   return new ExampleConnectorBOHandler();
}
```

Figure 58. The getConnectorBOHandlerForBO() method for generic business object handler

Creating a business object handler

Creating a business object handler involves the following steps:

- "Extending the Java business-object-handler base class"
- Implementing a business-object-handler retrieval method—For more information, see "Obtaining the Java business object handler" on page 153.
- "Implementing the doVerbFor() method" on page 155
- "Creating a custom business object handler" on page 174

Note: For an introduction to request processing, see "Request processing" on page 24.. For a discussion of request processing and the implementation of `doVerbFor()`, see Chapter 4, "Request processing," on page 79.

Extending the Java business-object-handler base class

In the Java connector library, the base class for a business object handler is named `CWConnectorBOHandler`. The `CWConnectorBOHandler` class provides methods for defining and accessing a business object handler. To implement your own business object handler, you extend this business-object-handler base class to create your own *business-object-handler class*.

Note: For general information about the methods of the business-object-handler base class, see "Extending the business-object-handler base class" on page 82..

To derive a business-object-handler class for a Java connector, follow these steps:

1. Create a class that extends the `CWConnectorBOHandler` class. Name this class:

   connectorName`BOHandler.java`

   where *connectorName* uniquely identifies the application or technology with which the connector communicates. For example, to create a business object handler for a Baan application, you create a business-object-handler class called `BaanBOHandler`. If your connector design implements multiple business object handlers, include the name of the handled business objects in the name of the business-object-handler class.
2. In the business-object-handler-class file, define the package name that contains your connector. A connector package name has the following format:

   `com.crossworlds.connectors.`*connectorName*

   where *connectorName* is the same as defined in step 1 above. For example, the package name for the Baan connector would be defined in the business-object-handler-class file as follows:

   ```
   package com.crossworlds.connectors.Baan;
   ```
3. Ensure that the business-object-handler-class file imports the following classes:

   ```
   com.crossworlds.cwconnectorapi.*;
   com.crossworlds.cwconnectorapi.exceptions.*;
   ```

   If you create several files to hold the business object handler's code, you must import these classes into *every* file.
4. Implement the `doVerbFor()` method to define the behavior of the business object handler. For more information on how to implement this method, see "Implementing the doVerbFor() method."

   Note: The other methods in the `CWConnectorBOHandler` class have their implementations provided. The `doVerbFor()` method is the only method you must implement in the business-object-handler class. For more information, see Chapter 12, "CWConnectorBOHandler class," on page 251.

You might need to implement more than one business object handler for your connector, depending on the application and its API. For a discussion of some issues to consider when implementing business object handlers, see "Designing business object handlers" on page 79..

Implementing the doVerbFor() method

The `doVerbFor()` method provides the functionality for the business object handler. When the connector framework receives a request business object, it calls the `doVerbFor()` method for the appropriate business object handler to perform the action of this business object's verb. For a Java connector, the `CWConnectorBOHandler` class defines the `doVerbFor()` method in which you define the verb processing.

Note: For a general description of the role of the `doVerbFor()` method, see "Handling the request" on page 82.. Figure 27 on page 83 provides the method's basic logic.

However, the actual doVerbFor() method that the connector framework invokes is the low-level version of this method, which the CWConnectorBOHandler class inherits from the BOHandlerBase class of the low-level Java connector library. This low-level version of doVerbFor() calls the user-implemented doVerbFor() method. Therefore, as part of your business-object-handler class (an extension of CWConnectorBOHandler), you must provide an implementation of the doVerbFor() method.

Note: The low-level doVerbFor() method calls the doVerbFor() method as long as the business object's verb does *not* contain the CBOH tag in its verb application-specific information. If the CBOH tag exists, the low-level doVerbFor() calls the custom business object handler whose name the CBOH tag specifies. For more information, see "Creating a custom business object handler" on page 174.

The role of the business object handler is to perform the following tasks:

1. Receive business objects from the connector framework.
2. Process each business object based on the active verb.
3. Send requests for operations to the application.
4. Return status to the connector framework.

Table 62 summarizes the steps in the basic logic for the verb processing that the doVerbFor() method typically performs. Each of the sections listed in the For More Information column provides more detailed information on the associated step in the basic logic.

Table 62. Basic logic of the doVerbFor() method

Business-object-handler step		For more information
1.	Obtain the active verb from the request business object.	"Obtaining the active verb" on page 157
2.	Verify that the connector still has a valid connection to the application.	"Verifying the connection before processing the verb" on page 158
3.	Branch on the value of the valid active verb.	"Branching on the active verb" on page 159
4.	For a given active verb, perform the appropriate request processing:	
	• Perform verb-specific tasks.	"Performing the verb operation" on page 161
	• Process the business object.	"Processing business objects" on page 162
5.	Send the appropriate status to the connector framework.	"Sending the verb-processing response" on page 169

In addition to the processing steps in Table 62,, this section also provides additional processing information in "Additional processing issues" on page 171.

Note: Java connectors must be thread safe. For Java connectors, the connector framework uses separate threads to call into the doVerbFor() and pollForEvents() methods.

WebSphere InterChange Server

If your business integration system uses InterChange Server and collaborations are coded to be multi-threaded, the connector framework might call into doVerbFor() with multiple threads representing request processing.

Obtaining the active verb

To determine which actions to take, the `doVerbFor()` method must first retrieve the verb from the business object that it receives as an argument. This incoming business object is called the *request business object*. The verb that this business object contains is the *active verb*, which must be one of the verbs that the business object definition supports. Table 63 lists the method that the Java connector library provides to retrieve the active verb from the request business object.

Table 63. Method for obtaining the active verb

Java connector library class	Method
`CWConnectorBusObj`	getVerb()

Obtaining the active verb from the request business object generally involves the following steps:

1. Verify that the request business object is valid.

   Before the connector calls `getVerb()`, it should verify that the incoming request business object is *not* `null`. The incoming business object is passed into the `doVerbFor()` method as a `CWConnectorBusObj` object.
2. Obtain the active verb with the `getVerb()` method.

   Once the request business object is valid, you can use the `getVerb()` method in the `CWConnectorBusObj` class to obtain the active verb from this business object.
3. Verify that the active verb is valid.

   When the connector has obtained the active verb, it should verify that this verb is neither null nor empty.

If *either* the request business object or the active verb is invalid, the connector should *not* continue with verb processing. Instead, it should take the steps outlined in Table 64..

Table 64. Handling a verb-processing error

Error-handling step	Method or code to use
1. Log an error message to the log destination to indicate the cause of the verb-processing error.	`CWConnectorUtil.generateAndLogMsg()`
2. Instantiate an exception-detail object to hold the exception information.	`CWConnectorExceptionObject` *`excptnDtailObj`* `= new CWConnectorExceptionObject();`
3. Set the status information within an exception-detail object:	
• set a message to indicate the cause of the verb-processing failure	*`excptnDtailObj`*`.setMsg()`
• set the status to the `FAIL` outcome status, which the connector framework includes in its response to the integration broker.	*`excptnDtailObj`*`.setStatus()`
4. Throw a `VerbProcessingFailureException` exception, which the `doVerbFor()` uses to tell the connector framework that a verb-processing error has occurred. This exception object contains the exception-detail object you initialized in Step 2. When the low-level `doVerbFor()` method catches this exception object, it copies the message and status from the exception-detail object into the return-status descriptor that it returns to the connector framework, which in turn returns it to the integration broker.	`throw new VerbProcessingFailureException(` *`excptnDtailObj`*`);`

Note: For information on the exception and exception-detail objects, see "Exceptions" on page 204.

Figure 59 contains a fragment of the `doVerbFor()` method that obtains the active verb with the `getVerb()` method. This code uses the `try` and `catch` statements to ensure that the request business object and its active verb are not null. If either of these conditions exists, the code fragment throws the `VerbProcessingFailedException` exception, which the connector framework catches.

```
public int doVerbFor(CWConnectorBusObj theBusObj)
   throws VerbProcessingFailedException, ConnectionFailureException
{
   CWConnectorExceptionObject cwExcpObj =
      new CWConnectorExceptionObject();

   //make sure that the incoming business object is not null
   if (theBusObj == null) {
      CWConnectorUtil.logMsg(3456,
         CWConnectorLogAndTrace.XRD_ERROR);
      cwExcpObj.setMsg(
         "doVerbFor(): Invalid business object passed in");
      cwExcpObj.setStatus(CWConnectorConstant.FAIL);
      throw new VerbProcessingFailedException(cwExcpObj);
   }

   // obtain the active verb
   String busObjVerb = theBusObj.getVerb();

   // make sure the active verb is neither null nor empty
   if (busObjVerb == null || busObjVerb.equals("")){
      cwExcpObj.setMsgNumber(6548);
      cwExcpObj.setMsgType(CWConnectorLogAndTrace.XRD_ERROR);
      cwExcpObj.setMsg("doVerbFor: Invalid active verb");
      cwExcpObj.setStatus(CWConnectorConstant.FAIL);
      throw new VerbProcessingFailedException(cwExcpObj);
   }
   try {
      // perform verb processing here
      ...

   } catch (SampleException se) {
      throw new VerbProcessingFailedException(cwExcpObj);
}
```

Figure 59. Obtaining the active verb

Verifying the connection before processing the verb

When the `agentInit()` method in the connector class initializes the application-specific component, one of its most common tasks is to establish a connection to the application. The verb processing that `doVerbFor()` performs requires access to the application. Therefore, before the `doVerbFor()` method begins processing the verb, it should verify that the connector is still connected to the application. The way to perform this verification is application-specific. Consult your application documentation for more information.

A good design practice is to code the connector application-specific component so that it shuts down whenever the connection to the application is lost. If the connection has been lost, the connector should *not* continue with verb processing. Instead, it should take the following steps to notify the connector framework of the lost connection:

1. Log an error message to the log destination to indicate the cause of the error.

   The connector logs a fatal error message so that email notification is triggered if the `LogAtInterchangeEnd` connector configuration property is set to True.
2. Set the exception-detail object with:

• a message to indicate the cause of the connection failure	`CWConnectorExceptionObject.setMsg()`
• the status of the `APPRESPONSETIMEOUT` outcome status, which the connector framework includes in its response to the integration broker.	`CWConnectorExceptionObject.setStatus()`

   This exception-detail object is part of the exception object that `doVerbFor()` throws. For information on these methods, see "Exceptions" on page 204.
3. Throw a `ConnectionFailureException` exception, which the `doVerbFor()` uses to tell the connector framework that a verb-processing cannot continue because the connection to the application has been lost. This exception object contains the exception-detail object you initialized in step 2..

   When the low-level `doVerbFor()` method catches this exception object, it copies the message and status from the exception-detail object into the return-status descriptor that it returns to the connector framework. If you have not set the status in the `ConnectionFailureException` exception-detail object, the connector framework sets the status to `APPRESPONSETIMEOUT`. The connector framework includes this return-status descriptor as part of its response to the integration broker. The integration broker can check the return-status descriptor to determine that the application is not responding.

After it has sent the return-status descriptor, the connector framework stops the process in which the connector runs. A system administrator must fix the problem with the application and restart the connector to continue processing events and business object requests.

Branching on the active verb

The main task of verb processing is to ensure that the application performs the operation associated with the active verb. The action to take on the active verb depends on whether the `doVerbFor()` method has been designed as a basic method or a metadata-driven method:

- "Basic verb processing"
- "Metadata-driven verb processing" on page 161

Basic verb processing: For verb-processing that is *not* metadata-driven, you branch on the value of the active verb to perform the verb-specific processing. Your `doVerbFor()` method must handle *all* verbs that the business object supports.

Note: You can obtain a list of business object's supported verbs with the `getSupportedVerbs()` method of the `CWConnectorBusObj` class.

Table 65 shows the verb constants that the Java connector library provides for comparing with the active verb.

Table 65. The Java verb constants

Verb Constant	Active Verb
`VERB_CREATE`	Create
`VERB_RETRIEVE`	Retrieve
`VERB_UPDATE`	Update

Table 65. The Java verb constants (continued)

Verb Constant	Active Verb
VERB_DELETE	Delete
VERB_EXISTS	Exists
VERB_RETRIEVEBYCONTENT	RetrieveByContent

All verb constants in Table 65 are defined in the CWConnectorConstant class. If your connector handles additional verbs, IBM recommends that you define your own String constants as part of your extended CWConnectorBOHandler class.

Note: As part of the verb-branching logic, make sure you include a test for an invalid verb. If the request business object's active verb is *not* supported by the business object definition, the business object handler must take the appropriate recovery actions to indicate an error in verb processing. For a list of steps to handle a verb-processing error, see Table 64 on page 157..

Figure 60 shows a code fragment of doVerbFor() that branches off the active verb's value for the Create and Update verbs. For each verb your business object supports, you must provide a branch in this code.

```
// handle the Create verb
if(busObjVerb.equals(CWConnectorConstant.VERB_CREATE)){
   CWConnectorUtil.initAndValidateAttributes(theBusObj);
   status=doCreate(theBusObj);
   // where doCreate() inserts new row into Sample Apps database
   // using data from theBusObj
   }

// handle the Update verb
else if (objVerb.equals(CWConnectorConstant.VERB_UPDATE)){
   status=doUpdate(theBusObj);
   // where doUpdate() locates existing row and updates it with
   // information from theObj

// notify connector framework of invalid verb
} else {
   CWConnectorUtil.logMsg(3456, CWConnectorLogAndTrace.XRD_ERROR);
   cwExcpObj.setMsg("doVerbFor(): Invalid verb passed in");
   cwExcpObj.setStatus(CWConnectorConstant.FAIL);
   throw new VerbProcessing FailedException(cwExcpObj);
}
```

Figure 60. Branching on the active verb's value

The code fragment in Figure 60 is modularized; that is, it puts the actual processing of each supported verb into a separate *verb method*: doCreate() and doUpdate(). Each verb method should meet the following minimal guidelines:

- Define a CWConnectorBusObj parameter, so the verb method can receive the request business object, and possibly send this updated business object back to the calling method.
- Throw any verb-specific exceptions to notify the doVerbFor() method of any verb-processing errors it encountered.
- Return an outcome status, which doVerbFor() can then return to the connector framework.

This modular structure greatly simplifies the readability and maintainability of the doVerbFor() method.

Metadata-driven verb processing: For metadata-driven verb processing, the application-specific information for the verb contains metadata, which provides processing instructions for the request business object when that particular verb is active. Table 66 lists the method that the Java connector library provides to obtain application-specific information for the verb of a business object.

Table 66. Method for retrieving the verb's application-specific information

Java connector library class	Method
`CWConnectorBusObj`	getVerbAppText()

The following call to `getVerbAppText()` extracts the verb's application-specific information:

```
String verbAppInfo = theBusObj.getVerbAppText(busObjVerb);
```

The verb application-specific information can contain the name of the method to call to process the request business object for that particular verb. In this case, the `doVerbFor()` method does *not* need to branch off the value of the active verb because the processing information resides in the verb's application-specific information.

Note: Another use of verb application-specific information can be to specify the application's API method to call to update the application entity for the particular verb.

Performing the verb operation

Table 67 lists the standard verbs that a `doVerbFor()` method can implement, as well as an overview of how each verb operation processes the request business object. For more information on how to process business objects, see "Processing business objects" on page 162.

Table 67. Performing the verb operation

Verb	Use of request business object	For more information
Create	• Use any application-specific information in the business object definition to determine in which application structure to create the entity (for example, a database table). • Use any application-specific information for each attribute to determine in which application substructure to add the attribute values (for example, a database column). • Use attribute values as values to save in new application entity. If the application generates key values for the new entity, save the new key values in the request business object, which should then be included as part of the verb-processing response.	"Handling the Create verb" on page 86
Retrieve	• Use any application-specific information in the business object definition to determine from which application structure (for example, a database table) to retrieve the entity. • Use attribute key value (or values) to identify which application entity to retrieve. If the application finds the requested entity, save its values in the request business object's attributes. The request business object should then be included as part of the verb-processing response.	"Handling the Retrieve verb" on page 89

Table 67. Performing the verb operation (continued)

Verb	Use of request business object	For more information
Update	• Use any application-specific information of the business object definition to determine in which application structure (for example, a database table) to update the entity. • Use any application-specific information for each attribute to determine which application substructure to update with the attribute values (for example, a database column). • Use attribute key value (or values) to identity which application entity to update. • Use the attribute values as values to update the existing application entity. If the application is designed to create an entity if the one specified for update does not exist, save the new entity values in the request business object's attributes. The request business object should then be included as part of the verb-processing response.	"Handling the Update verb" on page 97
Delete	• Use any application-specific information in the business object definition to determine from which application structure (for example, a database table) to delete the entity. • Use attribute key value (or values) to identify which application entity to delete. The request business object should then be included as part of the verb-processing response so that InterChange Server can perform any required cleanup of relationship tables.	"Handling the Delete verb" on page 104

Processing business objects

Most verb operations involve obtaining information from the request business object. This section provides information about the steps your `doVerbFor()` method needs to take to process the request business object.

Note: These steps assume that your connector has been designed to be metadata-driven; that is, they describe how to extract application-specific information from the business object definition and attributes to obtain the location within the application associated with each attribute. If your connector is *not* metadata-driven, you probably do not need to perform any steps that extract application-specific information.

Table 68 summarizes the steps in the basic program logic for deconstructing a request business object that contains metadata.

Table 68. Basic logic for processing a request business object with metadata

Step		For more information
1.	Obtain the business object definition for the request business object.	"Accessing the business object definition" on page 163
2.	Obtain the application-specific information in the business object definition to obtain the application structure to access.	"Extracting business object application-specific information" on page 163
3.	Obtain the attribute information.	"Accessing the attributes" on page 164
4.	For each attribute, get the attribute application-specific information in the business object definition to obtain the application substructure to access.	"Extracting attribute application-specific information" on page 165
5.	Make sure that processing occurs only for those attributes that are appropriate.	"Determining whether to process an attribute" on page 166

Table 68. Basic logic for processing a request business object with metadata (continued)

Step		For more information
6.	Obtain the value of each attribute whose value needs to be sent to the application entity.	"Extracting attribute values from a business object" on page 167
7.	Notify the application to perform the appropriate verb operation.	"Initiating the application operation" on page 168
8.	Save any attribute values in the request business object that are required for the verb-processing response.	"Saving attribute values in a business object" on page 168

Accessing the business object definition: For a Java connector, the `doVerbFor()` method receives the request business object as an instance of the `CWConnectorBusObj` class. To begin verb processing, the `doVerbFor()` method often needs information from the business object definition. The `CWConnectorBusObj` class provides access to the business object, its business object definition, and its attributes. Therefore, a Java `doVerbFor()` method does *not* need to instantiate a separate object for the business object definition; it can obtain information in the business object definition directly from the `CWConnectorBusObj` object passed into `doVerbFor()`.

The business object definition includes the information shown in Table 69.. For a complete list of `CWConnectorBusObj` methods, see Chapter 13, "CWConnectorBusObj class," on page 257.

Table 69. Methods for obtaining information from the business object definition

Business object definition information	CWConnectorBusObj method
The name of the business object definition	getName()
A verb list—contains the verbs that the business object supports	isVerbSupported(), getSupportedVerbs()
A list of attributes—for each attribute, the business object definition defines:	getAttrCount()
• attribute name	getAttrName()
• attribute data type	getTypeName(), getTypeNum()
• position in the list of attributes	getAttrIndex()
• other properties	For a complete list, see Table 71 on page 165..
Application-specific information:	
• business object definition	getAppText(), getBusObjASIHashtable()
• attribute	getAppText(), getAttrASIHashtable()
• verb	getVerbAppText()

A business object handler typically uses the business object definition to get information on the business object's attributes or to get the application-specific information from the business object definition, attribute, or verb.

Extracting business object application-specific information: Business objects for metadata-driven connectors are usually designed to have application-specific information that provides information about the application structure. For such connectors, the first step in a typical verb operation is to retrieve the application-specific information from the business object definition associated with the request business object. Table 70 lists the methods that the Java connector library provides to retrieve application-specific information from the business object definition.

Table 70. Methods for obtaining business object application-specific information

Java connector library class	Method
CWConnectorBusObj	getAppText() (with no arguments) getBusObjASIHashtable()

As Table 70 shows, the connector can use either of the following methods to obtain the application-specific information for the business object definition:

- The `getAppText()` method returns the application-specific information as a Java `String`. It can also retrieve the value of a specified name-value pair within the business-object-level application-specific information.

  Note: The `getAppText()` method uses deprecated terminology in its method name. This method name refers to "application-specific text". The more current name for "application-specific text" is "application-specific information".
- The `getBusObjASIHashtable()` method returns the application-specific information as a Java `Hashtable` of name-value pairs.

For a table-based application, the business objects are often designed to have application-specific information provide the verb operations with information about the application structure (For more information, see Table 43 on page 109). The application-specific information in a business object definition can contain the name of the database table associated with the business object.

Accessing the attributes: For a Java connector, the `CWConnectorBusObj` class provides access to the business object, its business object definition, *and* its attributes. When the `doVerbFor()` method needs information about attributes in the business object, it can obtain this information directly from the request business object. Therefore, a Java `doVerbFor()` method does *not* need to instantiate a separate object for an attribute.

The connector can use attribute methods in the `CWConnectorBusObj` class (see Table 71) to obtain information about an attribute, such as its cardinality or maximum length. Methods that access attribute properties provide the ability to access an attribute in of two ways:

- Its attribute name—you can identify the attribute by its Name property to obtain its attribute object:
- Its integer index—to obtain the attribute index (its ordinal position), you can either:
  - Obtain a count of all attributes in the business object definition with `getAttrCount()` and loop through them one at a time, passing each index value to one of the attribute-access methods in Table 71..
  - Obtain the index for a particular attribute. You can obtain the index for an attribute by specifying its name to `getAttrIndex()`.

  Note: Both the `getAttrCount()` and `getAttrIndex()` methods are defined in the `CWConnectorBusObj` class.

Table 71 lists the methods that the Java connector library provides to retrieve information about an attribute. For a complete list of methods that access attribute information, see Chapter 13, "CWConnectorBusObj class," on page 257.

Table 71. Methods for obtaining attribute information

Attribute property	CWConnectorBusObj method
Name	getAttrName(), hasName()
Type	getTypeNum(), getTypeName(), hasType(), isObjectType(), isType()
Key	isKeyAttr()
Foreign key	isForeignKeyAttr()
Max Length	getMaxLength()
Required	isRequiredAttr()
Cardinality	getCardinality(), hasCardinality(), isMultipleCard()
Default Value	getDefault(), getDefaultboolean(), getDefaultdouble(), getDefaultfloat(), getDefaultint(), getDefaultlong(), getDefaultString()
Attribute application-specific information	getAppText()

Extracting attribute application-specific information: If business objects for metadata-driven connectors are designed to have application-specific information that provides information about the application structure, the next step after extracting the application-specific information from the business object definition is to extract the application-specific information from each attribute in the request business object. Table 72 lists the methods that the Java connector library provides to retrieve application-specific information from each attribute.

Table 72. Methods for obtaining attribute application-specific information

Java connector library class	Method
`CWConnectorBusObj`	getAttrCount() getAppText() (with the position or name of the attribute as an argument) getAttrASIHashtable()

As Table 72 shows, the connector can use either of the following methods to obtain the application-specific information for an attribute:

- The `getAppText()` method returns the application-specific information as a Java `String`. It can also retrieve the value of a specified name-value pair within the attribute application-specific information.

  Note: The `getAppText()` method uses deprecated terminology in its method names. This method name refers to "application-specific text". The more current name for "application-specific text" is "application-specific information".
- The `getAttrASIHashtable()` method returns the application-specific information as a Java `Hashtable` of name-value pairs.

If business objects have been designed to have application-specific information provide information for a table-based application, the application-specific information for the attribute can contain the name of the application table's column associated with this attribute (For more information, see Table 43 on page 109). After extracting the application-specific information from the business object definition, the next step is to determine what columns in the application table are associated with the attributes in the request business object.

A verb operation can call `getAppText()`, passing it the position or name of the attribute, to obtain the name of the column within the database table to access. To obtain the application-specific information for *each* attribute, the verb operation must loop through *all* attributes in the business object definition. Therefore, it must determine the total number of attributes in the business object definition. The most common syntax for looping through the attributes is a `for` statement that uses the following limits on the loop index:

- Loop index is initialized to zero.

  If the verb operation processes the first attribute, which contains the key, the loop index variable starts at 0. However, if the verb is Create and your application generates keys, your Create verb operation should *not* process attributes containing keys. In this case, the loop index variable starts at a value other than 0.
- Loop index increments until it reaches the total number of attributes in the business object definition.

  The `getAttrCount()` method returns the total number of attributes in the business object. However, this total includes the `ObjectEventId` attribute. Because the `ObjectEventId` attribute is used by the IBM WebSphere business integration system and is *not* present in application tables, a verb operation does not need to process this attribute. Therefore, when looping through business object attributes, you loop from zero to one less than the total number of attributes:

  ```
  getAttrCount() - 1
  ```
- Loop index increments by one.

  This increment of the index obtains the next attribute.

Within the `for` loop, the Java connector can use the `getAppText()` method to obtain each attribute's application-specific information:

```
for (i = 0; i < theBusObj.getAttrCount() - 1; i++) {
   colName = theBusObj.getAppText(i);

   // process the attribute associated with the column in
   // 'colName'
}
```

Determining whether to process an attribute: Up to this point, the verb processing has been using the application-specific information to obtain the application location for each attribute of the request business object. With this location information, the verb operation can begin processing the attribute.

As the verb operation loops through the business object attributes, you might need to confirm that the operation processes only certain attributes. Table 73 lists some of the methods that the Java connector library provides to determine whether an attribute should be processed.

Table 73. Methods for determining attribute processing

Attribute test	CWConnectorBusObj method
An attribute is a simple attribute and *not* an attribute that represents a contained business object.	isObjectType()
The value of the attribute is *not* the special value of Blank (a zero-length string) or Ignore (a null pointer).	isIgnore(), isBlank()
The attribute is *not* a place-holder attribute. Place-holder attributes are used in business object definitions to separate attributes that contain child business objects.	getAppText()

Using the methods in Table 73,, a verb operation can determine that an attribute is one that the operation wants to process:

- Is the attribute simple or complex?

  The `isObjectType()` method checks that the attribute value does *not* represent a contained business object. For more information on how to handle an attribute that *does* contain a business object, see "Accessing child business objects" on page 173.
- Is the attribute a place-holder attribute or the `ObjectEventId` attribute?

  You can use the `getAppText()` method to determine if the attribute in the business object definition has application-specific information. Because neither of these special types of attributes represent columns in an application entity, there is no need for the business object definition to include application-specific information for them.
- Is the attribute set to a value other than the special Blank or Ignore values?

  The verb operation can compare the attribute's value to the Ignore and Blank values with the `isIgnore()` and `isBlank()` methods, respectively. For more information on the Ignore and Blank values, see "Handling the Blank and Ignore values" on page 171.

Extracting attribute values from a business object: Once the verb operation has confirmed that the attribute is ready for processing, it usually needs to extract the attribute value:

- For a Create or Update verb, the verb operation needs the attribute value to send it to the application, where it can be added to the appropriate application entity. For an Update verb, the verb operation also needs the attribute value from any key attribute that holds search information. The application uses this search information to locate the entity to update.

  Note: If the Create or Update operation sends information back to the connector, the verb operation needs to store the returned information as values in the appropriate attributes. For more information, see "Saving attribute values in a business object" on page 168.
- For a Retrieve, RetrieveByContent, or Exists verb, the verb operation needs the attribute value from any key attribute (Retrieve or Exists) or non-key attribute (RetrieveByContent) that holds search information. The application uses this search information to retrieve the entity.

  Note: For a Retrieve or RetrieveByContent, the verb operation also needs to set the attribute value for any attribute associated with retrieved data. For more information, see "Saving attribute values in a business object" on page 168.
- For a Delete verb, the verb operation needs the attribute value from any key attribute that holds search information. The application uses this search information to locate the entity to delete.

Table 74 lists the methods that the Java connector library provides to obtain attribute values from a business object.

Table 74. Methods for obtaining attribute values

Java connector library class	Method
CWConnectorBusObj	getTypeName(),, getTypeNum(),, getbooleanValue(),, getBusObjValue(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, getStringValue()

As Table 74 shows, the CWConnectorBusObj class provides type-specific methods for obtaining attribute values. These methods remove the need to cast the attribute value to match its type. You can choose which type-specific method to use by checking the attribute's data type with the getTypeName() or getTypeNum() method.

Initiating the application operation: Once the verb operation has obtained the information it needs from the request business object, it is ready to send the application-specific command so that the application performs the appropriate operation. The command must be appropriate for the verb of the request business object. For a table-based application, this command might be an SQL statement or a JDBC call. Consult your application documentation for more information.

Important: Your doVerbFor() method must ensure that the application operation completes successfully. If this operation is unsuccessful, the doVerbFor() method must return the appropriate outcome status (such as FAIL) to the connector framework. For more information, see "Sending the verb-processing response" on page 169.

Saving attribute values in a business object: Once the application operation has completed successfully, the verb operation might need to save new attribute values retrieved from the application into the request business object:

- For a Create verb, the verb operation needs to save the new key values if the application has generated them as part of its Create operation.
- For an Update verb, the verb operation needs to save *all* attribute values, including any generated key values (if the application has been designed to create a new entity when it does not find the specified entity to update).
- For a Retrieve or RetrieveByContent, the verb operation needs to save the attribute value for any attributes retrieved.

Table 75 lists the methods that the Java connector library provides to save attribute values in a business object.

Table 75. Methods for saving attribute values

Java connector library class	Method
CWConnectorBusObj	setAttrValues(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

As Table 75 shows, the CWConnectorBusObj class provides the following ways to save attribute values:

- The setAttrValues() method saves values for *all* attributes in a business object. It accepts the attribute values in a Java Vector object.
- The remaining methods in Table 75 are type-specific methods for saving attribute values. These methods remove the need to cast the attribute value to match its

type. You can choose which type-specific method to use by checking the attribute's data type with the `getTypeName()` or `getTypeNum()` method.

Sending the verb-processing response

The Java connector must send a verb-processing response to the connector framework, which in turn sends a response to the integration broker. This verb-processing response includes the following information:

- The integer return code of `doVerbFor()`
- A message in the return-status descriptor, if there is an information, warning, or error return message
- A response business object

The following sections provide additional information about how a Java connector provides each of the pieces of response information. For general information about the connector response, see "Indicating the connector response" on page 113..

Returning the outcome status: The `doVerbFor()` method provides an integer outcome status as its return code. As Table 76 shows, the Java connector library provides constants for the outcome-status values that `doVerbFor()` is mostly likely to return.

Important: The `doVerbFor()` method *must* return an integer outcome status to the connector framework.

Table 76. Outcome-status values for a Java doVerbFor()

Condition in doVerbFor()	Java outcome status
The verb operation succeeded.	`CWConnectorConstant.SUCCEED`
The verb operation failed.	`CWConnectorConstant.FAIL`
The application is not responding.	`CWConnectorConstant. APPRESPONSETIMEOUT`
At least one value in the business object changed.	`CWConnectorConstant.VALCHANGE`
The requested operation found multiple records for the same key value.	`CWConnectorConstant.VALDUPES`
The connector finds multiple matching records when retrieving using non-key values. The connector will only return the first matching record in a business object.	`CWConnectorConstant.MULTIPLE_HITS`
The connector was not able to find matches for Retrieve by non-key values.	`CWConnectorConstant. RETRIEVEBYCONTENT_FAILED`
The requested business object entity does not exist in the database.	`CWConnectorConstant. BO_DOES_NOT_EXIST`

Note: The `CWConnectorConstant` class provides additional outcome-status constants for use by other connector methods. For a complete list of outcome-status constants, see "Outcome-status constants" on page 305..

The outcome status that `doVerbFor()` returns depends on the particular active verb it is processing. Table 77 lists the tables in this manual that provide possible return values for the different verbs.

Table 77. Return values for different verbs

Verb	For more information
Create	Table 35 on page 88
Retrieve	Table 36 on page 94
RetrieveByContent	Table 37 on page 96
Update	Table 39 on page 103
Delete	Table 41 on page 105

Table 77. Return values for different verbs (continued)

Verb	For more information
Exists	Table 42 on page 106

Using the outcome status that `doVerbFor()` returns, the connector framework determines its next action:

- If the outcome status is `APPRESPONSETIMEOUT`, the connector framework shuts down the connector. For more information, see "Verifying the connection before processing the verb" on page 158.
- For all other outcome-status values, the connector framework continues running the connector. It copies the outcome status in its response to the integration broker. For some outcome-status values, the connector framework also includes a response business object in its response. For more information, see "Updating the request business object" on page 170.

Populating the return-status descriptor: The *return-status descriptor* is a structure that holds additional information about the state of the verb processing. When the connector framework invokes a business object handler, it actually calls the low-level version of the `doVerbFor()` method, inherited from the `BOHandlerBase` class of the low-level Java connector library. To this low-level `doVerbFor()` method, the connector framework passes in an empty return-status descriptor as an argument. The low-level `doVerbFor()` then calls the user-implemented `doVerbFor()` method, which is the version for which the connector developer provides an implementation as part of the `CWConnectorBOHandler` business-object-handler class. The user-implemented `doVerbFor()` performs the actual verb processing.

When this user-implemented `doVerbFor()` method exits, the low-level `doVerbFor()` updates its return-status descriptor with status information about the verb processing, as follows:

- If the user-implemented `doVerbFor()` method is successful (that is, it does *not* throw an exception), the low-level `doVerbFor()` copies the outcome status that the user-implemented `doVerbFor()` method returned into the status field of its return-status descriptor.
- If the user-implemented `doVerbFor()` method is *not* successful (that is, it throws one of the defined exceptions), the low-level `doVerbFor()` catches the exception and copies the status and any message from the exception-detail object into its return-status descriptor.

When the low-level `doVerbFor()` exits, this updated return-status descriptor is accessible by the connector framework. The connector framework then includes the return-status descriptor in the response it sends to the integration broker.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework returns the response to the connector controller, which routes it to the collaboration. This response includes the return-status descriptor populated by the low-level `doVerbFor()` method. The collaboration can access the information in this return-status descriptor to obtain the status of its service call request.

Updating the request business object: The connector framework passes in the request business object as an argument to `doVerbFor()`. The `doVerbFor()` method

can update this business object with attribute values. This updated business object is then accessible by the connector framework when `doVerbFor()` exits.

The connector framework uses the outcome status to determine whether to return a business object as part of its response to the integration broker, as follows:

- If the connector framework receives one of the following outcome-status values, it includes the request business object as part of its response:
  - `VALCHANGE`
  - `MULTIPLE_HITS`

  If your `doVerbFor()` method returns one of these outcome-status values, make sure it updates the request business object with appropriate response information.
- For any other outcome-status value, the connector framework does *not* include the request business object in its response.

Important: The outcome status that the `doVerbFor()` method returns affects what the connector framework sends to the integration broker. If the value is `VALCHANGE` or `MULTIPLE_HITS`, the connector framework returns the request business object. You must ensure that the request business object is updated as appropriate for the returned outcome status.

Additional processing issues

This section provides information on the following issues related to processing a business object:

- "Handling the Blank and Ignore values"
- "Accessing child business objects" on page 173

Handling the Blank and Ignore values: In addition to a regular attribute value, simple attributes in business objects can have either of the special values shown in Table 78..

Table 78. Special attribute values for simple attributes

Special attribute value	Represents
Blank	An "empty" zero-length string value
Ignore	An attribute value that the connector should ignore

WebSphere InterChange Server

Important: If your business integration system uses InterChange Server, in the third-party maps, the string `CxIgnore` represents an Ignore value, and the string `CxBlank` represents a Blank value. These strings should be used *only* in maps. They should *not* be stored in business objects as attribute values because they are reserved keywords in the IBM WebSphere InterChange Server system.

The connector can call Java connector library methods to determine whether a business object attribute is set to a special value:

- Blank—to process attributes with the Blank value, a connector can use any of the methods shown in Table 79..

Table 79. Methods for determining if an attribute contains the Blank value

CWConnectorBusObj method	Description
`isBlank(attributeName)` `isBlank(position)`	Determines whether a specified attribute contains the Blank value.

When an attribute contains the Blank value, the `doVerbFor()` method should process the attributes as shown in Table 81..

- Ignore— to process attributes with the Ignore value, a connector can use any of the methods shown in Table 80..

Table 80. Methods for determining if an attribute contains the Ignore value

CWConnectorBusObj method	Description
`isIgnore(attributeName)` `isIgnore(position)`	Determines whether a specified attribute contains the Ignore value.

When attributes are set to the Ignore value, the connector should process the attributes as shown in Table 82..

Table 81. Processing actions for the Blank value

Verb	Processing action for Blank value
Create	Create the entity with an appropriate blank value for the attributes. The blank value might be configurable, or it might be specific to the application.
Update	Update the entity fields to "empty" for those attributes that are set to the Blank value.
Retrieve	If the attribute is a key or the connector is doing a retrieve by non-key values, retrieve an entity where this attribute is a zero-length string.
Delete	If the attribute is a key, delete an entity where this field is set to the Blank value.

Table 82. Processing actions for the Ignore value

Verb	Processing action for Ignore value
Create	If the attribute is not a key, do not set a value in the application for the attribute. For key attributes, if the application generates keys, the key attributes might be set to the Ignore value. In this case, create the entity, retrieve the application-generated keys, and return the keys to the integration broker. Note that if the application does not generate key values, then all key attributes are expected to have valid values.
Update	If the attribute is not a key, do not set a value in the application for the attribute.
Retrieve	Do not match for Retrieve operations based on an attribute set to Ignore.
Delete	Do not match for Delete operations based on an attribute set to Ignore.

When a connector creates a new business object, all attribute values are set to Ignore internally. A connector must set appropriate values for attributes, since all unset attribute values remain defined as Ignore. To set attribute values to the special Ignore or Blank values, you use the methods in Table 83 (defined in the `CWConnectorUtil` class) to obtain a special attribute value and then assign the results of these methods directly to the attribute.

Table 83. Methods for obtaining special attribute values

Special attribute value	CWConnectorUtil method
Blank value	`getBlankValue()`
Ignore value	`getIgnoreValue()`

Once a method in Table 83 retrieves the desired special attribute value, you can pass it to one of the "set" methods for the attribute value (see Table 75 on page 168), as the following code fragment shows:

```
attrName = theBusObj.getAttrName(i);
theBusObj.setdoubleValue(attrName,
   CWConnectorUtil.getIgnoreValue());
```

Accessing child business objects: As discussed in "Processing hierarchical business objects" on page 109,, a Java connector uses the methods of the Java connector library shown in Table 75 to access child business objects.

Table 84. Methods for accessing child business objects

Java connector library class	Method
`CWConnectorBusObj`	isObjectType(),, isMultipleCard(),, getObjectCount(),, getBusObjValue()
`CWConnectorAttrType`	`OBJECT` attribute-type constant

The verb processing in the `doVerbFor()` method uses the `isObjectType()` method to determine if the attribute contains a business object (its attribute type is set to the `OBJECT` attribute-type constant). When a verb operation finds an attribute that is a business object, the method checks the cardinality of the attribute using `isMultipleCard()`. Based on the results of `isMultipleCard()`, the method takes one of the following actions:

- If the attribute has single cardinality, the method can perform the requested operation on the single child business object.
- If an attribute has multiple cardinality, the Java connector can access the contents of the business object array through the `CWConnectorBusObj` object:
  - If the attribute is a business object, it contains a `CWConnectorBusObj` object with one business object.
  - If the attribute is an business object array, it contains a `CWConnectorBusObj` object containing all business objects in the array.

The Java verb method can access individual business objects by calling `CWConnectorBusObj.getObjectCount()` to get the number of child business objects in the array. As it iterates through the business object array, the verb method can get each individual child object within the business object array using the `CWConnectorBusObj.getBusObjValue(`*`index`*`)` method, where *`index`* is the array element index. This method returns a `CWConnectorBusObj` that contains the a child business object. Figure 61 shows the Java code to access child business objects.

```
// For all attributes in the business object
for (int i=0; i<theBusObj.getAttrCount()-1; i++ ){
   if ( theBusObj.isObjectType(i) ){
      // cardinality N
      if(theBusObj.isMultipleCard(i)){
         for (int i=0; i < theBusObj.getObjectCount(); i++) {
            CWConnectorBusinessObject childBusObj =
               theBusObj.getBusObjValue(i);
            status = doVerbMethod(childBusObj);
         } // end for i to getObjectCount()
      } else {
         // Cardinality 1 child
            CWConnectorBusObj childBusObj = null;
         childBusObj = theBusObj.getBusObjValue(i);
         status = doVerbMethod(childBusObj);
      } // end else 1 cardinality
   } // end isObjectType()
} // end for i to getAttCount()-1
```

Figure 61. Accessing child business objects in a Java connector

Creating a custom business object handler

The connector framework calls the `doVerbFor()` method in the `CWConnectorBOHandler` class (which implements the business object handler) for *all* verbs that a particular business object supports.Therefore, all verbs in a business object are processed in one standard way (although they can initiate different actions within the application). However, if your connector supports a business object that requires different processing for some particular verb, you can create a *custom business object handler* to handle that verb for the business object.

Creating a custom business object handler involves the following steps:

- "Creating the class for the custom business object handler"
- "Implementing the doVerbForCustom() method" on page 175
- "Adding the verb application-specific information" on page 176

Creating the class for the custom business object handler

To create a custom business object handler, you must create a class that implements the `CWCustomBOHandler` interface. The `CWCustomBOHandler` interface provides the `doVerbForCustom()` method, which you must implement to define a custom business object handler. Follow these steps to create a custom-business-object-handler class for a Java connector:

1. Create a class that implements the `CWCustomBOHandler` interface. A suggested name this class is:

   connectorName`CustomBOHandler`*verbName*`.java`

   where *connectorName* uniquely identifies the application or technology with which the connector communicates and *verbName* identifies the verb (or verbs) that this custom business object handler processes. For example, to create a custom business object handler for the Retrieve verb in a Baan application, you create a custom-business-object-handler class called `BaanCustomBOHandlerRetrieve`.
2. In the custom-business-object-handler-class file, define the package name that contains your connector. A connector package name has the following format:

   `com.crossworlds.connectors.`*connectorName*

where *connectorName* is the same as defined in step 1 above. For example, the package name for the Baan connector would be defined in the custom-business-object-handler-class file as follows:

```
package com.crossworlds.connectors.Baan;
```

3. Ensure that the custom-business-object-handler-class file imports the following classes:

```
    com.crossworlds.cwconnectorapi.*;
com.crossworlds.cwconnectorapi.exceptions.*;
```

   If you create several files to hold the business object handler's code, you must import these classes into *every* file.

4. Implement the `doVerbForCustom()` method to define the behavior of the business object handler. For more information on how to implement this method, see "Implementing the doVerbForCustom() method."

Implementing the doVerbForCustom() method

The `doVerbForCustom()` method provides the functionality for the custom business object handler. As discussed in "Implementing the doVerbFor() method" on page 155, the connector framework calls the low-level `doVerbFor()` method (defined in the `BOHandlerBase` class) for the appropriate business object handler when it receives a request business object. This low-level `doVerbFor()` method determines which business object handler to call as follows:

- If the business object's verb has the `CBOH` tag in its application-specific information, call the `doVerbForCustom()` method.

  The `CBOH` tag specifies the full name (including the package name) of the custom-business-object-handler class, which implements the `CWCustomBOHandlerInterface` interface and its `doVerbForCustom()` method. For more information on this class name, see the description of "Adding the verb application-specific information" on page 176.

  If the `CBOH` tag exists, the low-level `doVerbFor()` method tries to create a new instance of the class that this tag specifies. If this instantiation is successful, the low-level `doVerbFor()` calls the `doVerbForCustom()` method in this class.

- Otherwise, call the `doVerbFor()` method, which the connector developer must implement as part of the business object handler's `CWConnectorBOHandler` class. For more information, see "Implementing the doVerbFor() method" on page 155.

The implementation of the `doVerbForCustom()` method must handle the verb processing of the verb for which its class is specified. You can refer to "Implementing the doVerbFor() method" on page 155 for information on the verb processing that the `doVerbFor()` method usually provides. However, you must customize the behavior of `doVerbForCustom()` to meet the special processing needs of your business object's verb.

Note: Unlike the `doVerbFor()` method, the `doVerbForCustom()` method is not invoked directly by the connector framework. Instead, the connector framework invokes the low-level `doVerbFor()`, which in turn invokes `doVerbForCustom()`. Therefore, `doVerbForCustom()` cannot include calls to any methods in the `CWConnectorBOHandler` class.

The low-level `doVerbFor()` method handles return values and exceptions from `doVerbForCustom()` as follows:

- On successful completion of `doVerbForCustom()`, send the status back to the connector framework (as it does for the `doVerbFor()` method).

- If there is any problem with the instantiation of the custom business object handler, populate the return-status descriptor with this status and an error message that describes the cause, then return a FAIL outcome status to the connector framework.
- If doVerbForCustom() throws the VerbProcessingFailedException exception, copy the status set in the exception object into the return-status descriptor, then return this exception status to the connector framework.
- If doVerbForCustom() throws the ConnectionFailureException exception, determine if the exception object has its status set:
  - If so, copy the exception status into the return-status descriptor and return this status to the connector framework.
  - If not, copy the APPRESPONSETIMEOUT outcome status into the return-status description and return APPRESPONSETIMEOUT to the connector framework.

Adding the verb application-specific information

For the connector framework to call a custom business object handler for a particular business object, the verb of this business object must have the CBOH tag in its verb application-specific information. The CBOH tag has the following format:

CBOH=*connectorPackageName*.*CustomBOHandlerClassName*

In this format, the *connectorPackageName* is as follows:

com.crossworlds.connectors.*connectorName*

with *connectorName* the name of the connector. The *CustomBOHandlerClassName* is the name of the class that implements the CWCustomBOHandlerInterface interface.

For example, the following CBOH tag specifies a class called BaanCustomBOHandlerRetrieve:

```
CBOH=com.crossworlds.connectors.Baan.BaanCustomBOHandlerRetrieve
```

Implementing an event-notification mechanism

Table 85 shows the support that the Java connector library provides for the development of an event-notification mechanism:

Table 85. Support for an event-notification mechanism

Java connector library support	For more information
The following classes for the encapsulation of access to the event store: • CWConnectorEvent • CWConnectorEventStatusConstants • CWConnectorEventStore • CWConnectorEventStoreFactory	"Obtaining access to the event store" on page 177
A poll method, pollForEvents(), that polls the event store at a specified frequency.	"Implementing the pollForEvents() method" on page 180

Note: For an introduction to event notification, see "Event notification" on page 22.. For a discussion of event-notification mechanisms and the implementation of pollForEvents(), see Chapter 5, "Event notification," on page 115.

Obtaining access to the event store

If a connector is expected to process information that originates in its application, it must obtain access to the application's event store. Table 86 shows the support that the Java connector library provides in support of obtaining access to an event store from within a Java connector.

Table 86. Support for defining access to an event store

	Java connector library class	Description
Event store	`CWConnectorEventStoreFactory`	Provides a single method that creates an event-store object
	`CWConnectorEventStore`	Represents the event store
Event	`CWConnectorEvent`	Represents an event object, which provides access to an event record within the Java connector.

Defining the event store

As Table 86 shows, the Java connector library provides the following classes to define an event store:

- "CWConnectorEventStore class"
- "CWConnectorEventStoreFactory interface" on page 178

CWConnectorEventStore class: The `CWConnectorEventStore` class defines an event store. As Table 87 shows, this class provides an additional layer for standardizing the event retrieval, processing, and archiving mechanisms.

Table 87. Methods of the CWConnectorEventStore class

Event-store task	CWConnectorEventStore method	Implementation status
Event retrieval	fetchEvents()	Must be implemented
	getBO()	Implementation provided in base class—however, you must override this implementation if your connector does not support the RetrieveByContent verb.
	getNextEvent()	Implementation provided in base class
Event processing	recoverInProgressEvents()	Must be implemented
	resubmitArchivedEvents()	Must be implemented
	setEventStatus()	Must be implemented
	setEventsToProcess()	Implementation provided in base class
	updateEventStatus()	Implementation provided in base class
Archiving	archiveEvent()	Must be implemented—if the connector supports archiving.
	deleteEvent()	Must be implemented
Error processing	getTerminate(),, setTerminate()	Implementation provided in base class
Resource cleanup	cleanupResources()	Not required for the event-store class but must be implemented if resources used to access the event store need to be released.

To define an event store, follow these steps:

1. Extend the `CWConnectorEventStore` class, naming your new class to identify the event store that your connector accesses.
2. Define any additional data members that your event store might require.

   The `CWConnectorEventStore` class contains a single data member: an events vector array called `eventsToProcess`. Events retrieved from the event store are saved in this Java `Vector` object. Declare any other information that is required

to access the application's event and archive stores as data members in your extended CWConnectorEventStore class. This information should include the location of the event and archive stores. For example:

- In a table-based application, this information might be the event and archive table names and any database connection information.
- In a file-based event store, this information might include the names of the event and archive directories.
- An extended event store should also store any metadata information required for accessing or processing the event records. This information might include any "order by" information needed for JDBC queries

3. Implement the appropriate abstract methods within the CWConnectorEventStore class (see Table 87) to provide access to the event store.

   You can implement those CWConnectorEventStore methods that your event store requires, with the following conditions:

   - You *must* provide implementations for the abstract methods with "Must be implemented" in the Implementation Status column of Table 87.. These methods are required to support the default implementation of the pollForEvents() method.

     Note: If you override the default implementation of pollForEvents(), you can define only those CWConnectorEventStore methods that your pollForEvents() method needs to use.

   - The CWConnectorEventStore class provides implementations for the methods with "Implementation provided in base class" in the Implementation Status column of Table 87..

4. Access the CWConnectorEventStore methods as needed to perform event retrieval, event processing, and archiving from within the pollForEvents() poll method. For more information, see "Implementing the pollForEvents() method" on page 180.

Note: For more information on the methods of CWConnectorEventStore, see Chapter 17, "CWConnectorEventStore class," on page 319.

CWConnectorEventStoreFactory interface: The CWConnectorEventStoreFactory interface defines an event-store factory, which provides a method to instantiate an event store, as Table 88 shows.

Table 88. Method of the CWConnectorEventStoreFactory interface

CWConnectorEventStoreFactory method	Implementation status
getEventStore()	Must be implemented

To define an event-store factory, follow these steps:

1. Create a new event-store-factory class to implement the CWConnectorEventStoreFactory interface. Name your new class to include the name of the event store that your CWConnectorEventStore class accesses.
2. Implement the getEventStore() method of the CWConnectorEventStoreFactory interface within your event-store-factory class to provide an event-store factory for your extended CWConnectorEventStore class.
3. Determine whether to use the default implementation of the getEventStore() method in the CWConnectorAgent class to instantiate an event store. The default implementation of the pollForEvents() method uses this getEventStore() method to obtain a reference to the event store.

- If you use the default implementation of this `getEventStore()` method, define the `EventStoreFactory` connector configuration property and set it to the entire class name (including its package name) for your event-store-factory class (which implements the `CWConnectorEventStoreFactory` interface).

  The `EventStoreFactory` property has the following format:

  `connectorPackageName.EventStoreFactoryClassName`

  In this format, the *connectorPackageName* is as follows:

  `com.crossworlds.connectors.`*`connectorName`*

  with *connectorName* the name of the connector. The *EventStoreFactoryClassName* is the name of the class that implements the `CWConnectorEventStoreFactory` interface.

  Note: The `EventStoreFactory` property is a user-defined property, *not* a standard property. You must define this property with Connector Configurator for any connector that provides an event-store factory.

  If `EventStoreFactory` is *not* set, the default implementation of `getEventStore()` attempts to generate the name of the event store. For more information, see the description of "getEventStore()" on page 240..
- If the default implementation of `getEventStore()` does not adequately address the needs of your connector, you can override it in your connector class. Within this method, you can call some custom event-store constructor.

Defining an event object

The Java connector obtains event records from the event store and encapsulates them as *event objects*. The event-store class builds event objects for each event record that the connector retrieves from the event store. The information in each event object is then used to build and retrieve the business object that the connector sends to the integration broker.

The default event object that `CWConnectorEvent` defines contains the event information in Table 46 on page 116.. The `CWConnectorEvent` class provides access methods for this information, as Table 89 shows.

Table 89. Methods to retrieve information in an event object

Element	CWConnectorEvent method
Event Id	getEventID()
Business object name	getBusObjName()
Business object verb	getVerb()
Object key	getIDValues(),, getKeyDelimiter() These `CWConnectorEvent` methods provide access to the actual data values that identify the business object. The `getIDValues()` method assumes that this data is a name/value pair. For example, if the object key contains data for the `ContractId` attribute in the business object, the name/value pair in the business object data would be: `ContractId=45381`If the object key in the event record contains a concatenation of fields, the `getIDValues()` assumes that each name/value pair is separated by a delimiter, which the `getKeyDelimiter()` method returns. The delimiter should be configurable as set by the `PollAttributeDelimiter` connector configuration property. The default value for the delimiter is a colon (:).
Priority	getPriority()
Timestamp	getEventTimeStamp()

Table 89. Methods to retrieve information in an event object (continued)

Element	CWConnectorEvent method
Status	getStatus() Use the following methods to set event status: getNextEvent(),, recoverInProgressEvents(),, resubmitArchivedEvents(),, setEventStatus(), updateEventStatus().
Description	A text string describing the event.
ConnectorID	getConnectorID()

In addition to providing the standard information in an event record (shown in Table 89), the event object also provides accessor methods for the information shown in Table 90..

Table 90. Additional event information in the event object

Element	Description	Accessor method
Effective date	Date on which the event becomes active and should be processed. This information might be useful when there is a change to an object in one system that should *not* be propagated until the date on which it becomes effective (such as a salary change).	getEffectiveDate()
Event source	Source from where the event originated. This information might be needed by a connector that needs to track the event source for archiving.	getEventSource(),, setEventSource()
Triggering user	User identifier (ID) associated with the user that triggered this event. This information can be used to avoid synchronization problems between two systems.	getTriggeringUser()

If your event record requires information beyond what the default event class provides (Table 89 and Table 90), you can take the following steps:

1. Extend the `CWConnectorEvent` class, naming your new class to identify the event store whose event records your event class encapsulates.
2. Define any additional data members that your event might require.

   The `CWConnectorEvent` class contains the data members whose accessor methods are listed in Table 89 and Table 90.. Any other information that is required to access the application's event records needs to be declared as data members in your extended `CWConnectorEvent` class.
3. Provide accessor methods for any data members you add to your extended `CWConnectorEvent` class.

   To support true encapsulation, your data members should be private members of your extended `CWConnectorEvent` class. To provide access to these data members, you define a "get" methods to retrieve each data member's value. You can also define "set" methods for those data members that connector developers are allowed to set.

Note: For more information on the methods of `CWConnectorEvent`, see Chapter 15, "CWConnectorEvent class," on page 307.

Implementing the pollForEvents() method

For a Java connector, the `CWConnectorAgent` class defines the `pollForEvents()` method. This class provides a default implementation of `pollForEvents()`. You can use this default implementation or override the method with your own poll method. However, the `pollForEvents()` method must be implemented.

The Java-based pseudo-code in Figure 62 shows the basic logic flow for a `pollForEvents()` method. The method first retrieves a set of events from the event store. For each event, the method calls the `isSubscribed()` method to determine whether any subscriptions exist for the corresponding business object. If there are subscriptions, the method retrieves the data from the application, creates a new business object, and calls `gotApplEvent()` to send the business object to InterChange Server. If there are no subscriptions, the method archives the event record with a status value of unprocessed.

```
public int pollForEvents()
{
   int status = 0;
   get the events from the event store
   for (events 1 to MaxEvents in event store) {
      extract BOName, verb, and key from the event record
        if(ConnectorBase.isSubscribed(BOName,BOverb) {
            BO = JavaConnectorUtil.createBusinessObject(BOName)
            BO.setAttrValue(key)

            retrieve application data using doVerbFor()
            BO.setVerb(Retrieve)
            BO.doVerbFor()
            BO.setVerb(BOverb)
            status = gotApplEvent(BusinessObject);

            archive event record with success or failure status
        }
        else {
            archive item with unsubscribed status
   }
   return status;
}
```

Figure 62. Java pollForEvents() example

Note: For a flow chart of the poll method's basic logic, see Figure 27 on page 83..

This section provides more detailed information on each of the steps in the basic logic for the event processing that the `pollForEvents()` method typically performs. Table 91 summarizes these basic steps.

Table 91. Basic logic of the pollForEvents() method

Step		For more information
1.	Set up a subscription manager for the connector.	"Accessing a subscription manager" on page 182
2.	Verify that the connector still has a valid connection to the event store.	"Verifying the connection before accessing the event store" on page 182
3.	Retrieve specified number of event records from the event store and store them in an events array. Cycle through the events array. For each event, mark the event in the event store as In-Progress and begin processing.	"Retrieving event records" on page 182
4.	Get the business object name, verb, and key data from the event record.	"Getting the business object name, verb, and key" on page 184
5.	Check for subscriptions to the event.	"Checking for subscriptions to the event" on page 185
	If the event has subscribers: • Retrieve application data and create the business object.	"Retrieving application data" on page 187

Table 91. Basic logic of the pollForEvents() method (continued)

Step	For more information
• Send the business object to the connector framework for event delivery. • Complete event processing.	"Sending the business object to the connector framework" on page 188 "Completing the processing of an event" on page 192
If the event does *not* have subscribers, update the event status to Unsubscribed.	"Checking for subscriptions to the event" on page 185
6. Archive the event.	"Archiving the event" on page 193
7. Release resources used to access the event store.	

Accessing a subscription manager

As part of connector initialization, the connector framework instantiates a subscription manager. This subscription manager keeps the subscription list current. (For more information, see "Business object subscription and publishing" on page 13.) A connector has access to the subscription manager and the connector subscription list through a *subscription handler*, which is included in the connector base class. It can use methods of this class to determine whether business objects have subscribers and to send business objects to the connector controller.

Note: Unlike a C++ connector, a Java connector does *not* need to set up a subscription handler. This functionality is handled in the `CWConnectorAgent` class.

Verifying the connection before accessing the event store

When the `agentInit()` method in the connector class initializes the application-specific component, one of its most common tasks is to establish a connection to the application. The poll method requires access to the event store. Therefore, before the `pollForEvents()` method begins processing events, it should verify that the connector is still connected to the application. The way to perform this verification is application-specific. Consult your application documentation for more information.

A good design practice is to code the connector application-specific component so that it shuts down whenever the connection to the application is lost. If the connection has been lost, the connector should *not* continue with event polling. Instead, it should return `APPRESPONSETIMEOUT` to notify the connector framework of the loss of connection to the application.

Note: To surface an `APPRESPONSETIMEOUT` outcome status returned by the `doVerbFor()` from within `pollForEvents()`, use the `getTerminate()` method of the `CWConnectorEventStore` class. For more information, see "Retrieving application data" on page 187.

Retrieving event records

To send event notifications to the connector framework, the poll method must first retrieve event records from the event store. Table 92 lists the methods that the Java connector library provides to retrieve event records from the event store.

Table 92. Classes and methods for event retrieval

Java connector library class	Method
`CWConnectorAgent`	getEventStore()
`CWConnectorEventStoreFactory`	getEventStore()
`CWConnectorEventStore`	fetchEvents(),, `getNextEvent()`,,updateEventStatus()

The poll method can retrieve one event record at a time and process it or it can retrieve a specified number of event records per poll and cache them to an events array. Processing multiple events per poll can improve performance when the application generates large numbers of events.

The number of events picked up in any polling cycle should be configurable using the connector configuration property `PollQuantity`. At install time, a system administrator sets the value of `PollQuantity` to an appropriate number, such as 50. The poll method can use the `getConfigProp()` to retrieve the value of the `PollQuantity` property, and then retrieve the specified number of event records and process them in a single poll.

The connector should assign the In-Progress status to any event that it has read out of the event store and has started to process. If the connector terminates while processing an event and before updating the event status to indicate that the event was either sent or failed, it will leave an In-Progress event in the table. For more information on how recover these In-Progress events, see "Recovering In-Progress events" on page 151.

The Java connector library provides the `CWConnectorEventStore` class to represent an event store. To retrieve event records from this event store, the poll method takes the following actions:

1. Instantiate an event-store object with the `getEventStore()` method that is defined in the `CWConnectorAgent` class. The default implementation of this method calls the `getEventStore()` of the event-store-factory class named in the `EventStoreFactory` connector configuration property. The event-store-factory class implements the `CWConnectorEventStoreFactory` interface for your event store. For more information, see "CWConnectorEventStoreFactory interface" on page 178.
2. Retrieve a specified number of event records from the event store with the `fetchEvents()` method.

   You must implement the `fetchEvents()` method as part of the `CWConnectorEventStore` class. This method can use the value of the `PollQuantity` connector configuration property as the number of event records to retrieve. The method must take the following actions:

   - Create a `CWConnectorEvent` event object for each event record that it retrieves.

     These event records can be ordered by their timestamp. For information on retrieving event records by event priority, see "Processing events by event priority" on page 131..

     Note: If the event store is implemented with an event table in the application database, the `fetchEvents()` method can use JDBC methods to access the event table, in much the same way as the C++ connector uses ODBC methods.

   - Put each event object into the `eventsInProgress` events vector.

   The `fetchEvents()` method should throw the `StatusChangeFailedException` exception if the application is unable to fetch events because it is unable to access the event store. When the `pollForEvents()` method catches this exception, it can return the `APPRESPONSETIMEOUT` outcome status to indicate the lack of response from the application's event store.

- Loop through the events in the `eventsInProgress` events vector, taking the following actions on each event object:

- Retrieve the next event object to process with the `getNextEvent()` method.
- Update the status of both the event record (in the event store) and the event object (retrieved from the events vector) to `IN_PROGRESS` with the `updateEventStatus()` method.

  The `updateEventStatus()` method should throw the `StatusChangeFailedException` exception if the application is unable to change event status because it is unable to access the event store. When the `pollForEvents()` method catches this exception, it can return the `APPRESPONSETIMEOUT` outcome status to indicate the lack of response from the application's event store.

Setting the event status to `IN_PROGRESS` indicates that the poll method has begun processing on the event. Figure 63 shows a code fragment that retrieves event records from the event store, accessing each as an event object.

```
// Instantiate event store
CWConnectorEventStore evts=getEventStore();

// Fetch PollQuantity number of events from the application.
try
   {
   evts.fetchEvents();
   }
catch (StatusChangeFailedException e)
   {
   // log error message
   return CWConnectorConstant.FAIL;
   }
}
// Get the property values for PollQuantity
int pollQuantity;
String poll=CWConnectorUtil.getConfigProp("PollQuantity");
if (poll == null || poll.equals(""))
   pollQuantity=1;
else
   pollQuantity=Integer.parseInt(poll);

   for (int i=0; i < pollQuantity; i++)
   {
      // Process each event retrieved from the application.
      // Get the next event to be processed.
      evtObj=evts.getNextEvent();
```

Figure 63. Retrieving event records from the event store

Getting the business object name, verb, and key

Once the connector has retrieved an event, it extracts the event ID, the object key, and the name and verb of the business object from the event record. The connector uses the business object name and verb to determine whether the integration broker is interested in this type of business object. If the business object and its active verb have subscribers, the connector uses the entity key to retrieve the complete set of data.

Table 93 lists the methods that the Java connector library provides to obtain the name of the business object definition and the verb from the retrieved event records.

Table 93. Methods for obtaining event information

Java connector library class	Method
`CWConnectorEvent`	getBusObjName(),, getVerb()

Important: The connector should send the business object with the same verb that was in the event record.

Once the `getNextEvent()` method has retrieved an event object to be processed, the Java connector can use the appropriate accessor methods of the `CWConnectorEvent` class to obtain the information needed to check for an event subscription, as follows:

Event ID	getEventID()
Business object name	getBusObjName()
Verb	getVerb()
Object key	getIDValues()

For sample code that uses these accessor methods, see Figure 64 on page 186..

Checking for subscriptions to the event

To determine whether the integration broker is interested in receiving a particular business object and verb, the poll method calls the `isSubscribed()` method. The `isSubscribed()` method takes the name of the current business object and a verb as arguments. The name of the business object and verb must match the name of the business object and verb in the repository.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the poll method can determine if any collaboration subscribes to the business object with a particular verb. At initialization, the connector framework requests its subscription list from the connector controller at connector initialization. At runtime, the application-specific component can use `isSubscribed()` to query the connector framework to verify that some collaboration subscribes to a particular business object. The application-specific connector component can send the event only if some collaboration is currently subscribed.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework assumes that the integration broker is interested in *all* the connector's supported business objects. If the poll method uses the `isSubscribed()` method to query the connector framework about subscriptions for a particular business object, the method returns `true` for *every* business object that the connector supports.

Table 94 lists the methods that the Java connector library provides to check for subscriptions to the event.

Table 94. Classes and methods for checking subscriptions

Java connector library class	Method
CWConnectorAgent	isSubscribed()
CWConnectorEventStore	updateEventStatus(),, archiveEvent(),, deleteEvent()

Based on the value that isSubscribed() returns, the poll method should take one of the following actions based on whether there are subscribers for the event:

- If there are subscribers for an event, the connector takes one of the actions described in "Events that have subscriptions."
- If there are no subscriptions for the event, the connector should take one of the actions described in "Events that do not have subscriptions."

For a Java connector, the isSubscribed() method is defined in the CWConnectorAgent class because the subscription manager is part of the connector base class. The method returns true if there are subscribers and false if there are no subscribers. Figure 64 shows a code fragment that checks for subscriptions in a Java connector.

```
if (isSubscribed(evtObj.getBusObjName(),evtObj.getVerb())) {
      // handle event
} else
   {
      // Update the event status to UNSUBSCRIBED.
      evts.updateEventStatus(evtObj,
         CWConnectorEventStatusConstants.UNSUBSCRIBED);

      // Archive the event (if archiving is supported)
      return CWConnectorConstant.FAIL;
   }
```

Figure 64. Checking for an event subscription

If no subscriptions exist for the event, this code fragment uses the updateEventStatus() method to update the event's status to UNSUBSCRIBED and then archives the event.

Events that have subscriptions: If there are subscribers for an event, the connector takes the following actions:

Connector action taken	For more information
Retrieve the complete set of business object data from the entity in the application database.	"Retrieving application data" on page 187
Send the business object to the connector framework, which routes it to the integration broker.	"Sending the business object to the connector framework" on page 188
Complete the processing on the event.	"Completing the processing of an event" on page 192
Archive the event (if archiving is implemented) in case the integration broker subscribes at a later time.	"Archiving the event" on page 193

Events that do not have subscriptions: If there are no subscriptions for the event, the connector should take the following actions:

- Update the status of the event to "Unsubscribed" to indicate that there were no subscribers.
- Archive the event (if archiving is implemented) in case the integration broker subscribes at a later time. Moving the event record to the archive store prevents the poll method from picking up unsubscribed events. For more information, see "Archiving the event" on page 193.
- Return "fail" (`FAIL` outcome status for a Java connector) to indicate there are events pending for which no subscriptions currently exist.

  IBM suggests that the connector return "fail" if no subscriptions exist for the event. However, you can return the outcome status that your design dictates.

No other processing should be done with unsubscribed events. If at a later date, the integration broker subscribes to these events, a system administrator can move the unsubscribed event records from the archive store back to the event store.

Retrieving application data

If there are subscribers for an event, the poll method must take the following steps:

1. Retrieve the complete set of data for the entity from the application.

   To retrieve the complete set of entity data, the poll method must use name of the entity's key information (which is stored in the event) to locate the entity in the application. The poll method must retrieve the complete set of application data when the event has the following verbs:

   - Create
   - Update
   - Delete event for an application that supports logical deletes

   For a Delete event from an application that supports physical deletes, the application may have already deleted the entity from the database, and the connector may not be able to retrieve the entity data. For information on delete processing, see "Processing Delete events" on page 132.
2. Package the entity data in a business object.

   Once the populated business object exists, the poll method can publish the business object to subscribers.

Table 95 lists the method that the Java connector library provides to retrieve entity data from the application database and populate a business object.

Table 95. Method for retrieving business object data

Java connector library class	Method
`CWConnectorEventStore`	getBO()

Note: If the event is a delete operation and the application supports physical deletions of data, the data has most likely been deleted from the application, and the connector cannot retrieve the data. In this case, the connector simply creates a business object, sets the key from the object key of the event record, and sends the business object.

For a Java connector, the standard way of retrieving application data from within `pollForEvents()` is to use the `getBO()` method in the `CWConnectorEventStore` class. This method takes the following steps:

- Create a temporary `CWConnectorBusObj` object to hold the new business object.
- Populate the `CWConnectorBusObj` object with the data and key values from the specified event object.

- If the event's verb is Create or Update, set the business object's verb to RetrieveByContent and call the doVerbFor() method to retrieve the remaining attribute values from the application.
- Return the populated CWConnectorBusObj object to the caller.

If the call to getBO() is successful, it returns the populated CWConnectorBusObj object. The following line shows a call to getBO() that returns a populated CWConnectorBusObj object called bo:

```
bo = evts.getBO(evtObj);
```

In case the getBO() call is *not* successful, the poll method should take the following steps:

- Catch any exceptions that getBO() throws.
- Check for an ERROR_OBJECT_NOT_FOUND status in the event object to determine if the doVerbFor() method could not find the business object data in the application.
- Check for a null value returned by getBO(), which indicates that doVerbFor() was not successful.
- Use the getTerminate() method to check if the terminate-connector flag has been set, which indicates that doVerbFor() (called from within the getBO() method) returned an APPRESPONSETIMEOUT outcome status. If getTerminate() returns true, pollForEvents() should return an APPRESPONSETIMEOUT outcome status to terminate the connector.

  Note: The default implementation of getBO() checks the outcome status of doVerbFor() and calls the setTerminate() method if doVerbFor() returns an APPRESPONSETIMEOUT outcome status. If you override the default implementation of getBO() but still use the default implementation of pollForEvents(), your getBO() implementation should perform this same task.

The ObjectEventId attribute is used in the IBM WebSphere business integration system to track the flow of business objects through the system. In addition, it is used to keep track of child business objects across requests and responses, as child business objects in a hierarchical business object request might be reordered in a response business object.

Connectors are not required to populate ObjectEventId attributes for either a parent business object or its children. If business objects do not have values for ObjectEventId attributes, the business integration system generates values for them. However, if a connector populates child ObjectEventIds, the values must be unique across all other ObjectEventId values for that particular business object regardless of level of hierarchy. ObjectEventId values can be generated as part of the event notification mechanism. For suggestions on how to generate ObjectEventId values, see "Event identifier" on page 117.

Sending the business object to the connector framework

Once the data for the business object has been retrieved, the poll method performs the following tasks:

- "Setting the business object verb" on page 189
- "Sending the business object" on page 189

Table 96 lists the methods that the Java connector library provides to perform these tasks.

Table 96. Classes and methods for setting the verb and sending the business object

Java connector library class	Method
CWConnectorBusObj	setVerb()
CWConnectorEvent	getVerb()
CWConnectorAgent	gotApplEvent()

Setting the business object verb: To set the verb in a business object to the verb specified in the event record, the poll method calls the business object method `setVerb()`. The poll method should set the verb to the same verb that was in the event record in the event store.

Note: If the event is a physical delete, use the object keys from the event record to set the keys in the business object, and set the verb to Delete.

For a Java connector, the populated `CWConnectorBusObj` object that the `getBO()` method returns still has a verb of RetrieveByContent. The poll method must set the business object's verb to its original value with the `setVerb()` method of the `CWConnectorBusObj` class, as the following code fragment shows:

```
// Set verb to action as indicated in the event record
busObj.setVerb(evntObj.getVerb());
```

In this code fragment, the poll method uses the `getVerb()` of the `CWConnectorEvent` class to obtain the verb from the event record. This verb is then copied into the business object with `setVerb()`.

Sending the business object: The poll method uses the method `gotApplEvent()` to send the business object to the connector framework. This method takes the following steps:

- Check that the connector is active.
- Check that there are subscriptions for the event.
- Send the business object to the connector framework.

The connector framework does some processing on the event object to serialize the data and ensure that it is persisted properly. It then makes sure the event is sent.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework makes sure the event is either sent to the ICS through CORBA IIOP or written to a queue (if you are using queues for event notification). If sending the event to ICS, the connector framework forwards the business object to the connector controller, which in turn performs any mapping required to transform the application-specific business object to a generic business object. The connector controller can then send the generic business object to the appropriate collaboration.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework makes sure the event is converted to an XML WebSphere MQ message and written to the appropriate MQ queue.

The poll method should check the return code from `gotApplEvent()` to ensure that any error conditions are handled appropriately. For example, until the event delivery is successful, the poll method should *not* remove the event from the event store. Instead, the poll method should update the event record's status to reflect the results of the event delivery. Table 97 shows the possible event-status values, based on the return code from `gotApplEvent()`.

Table 97. Possible event status after event delivery with gotApplEvent()

State of event delivery	Return code of gotApplEvent()	Event status
If the event delivery is successful	`SUCCEED`	`SUCCESS`
If no subscription exists for the event	`NO_SUBSCRIPTION_FOUND`	`UNSUBSCRIBED`
If the connector has been paused	`CONNECTOR_NOT_ACTIVE`	`READY_FOR_POLL`
If the event delivery fails	`FAIL`	`ERROR_POSTING_EVENT`

The `gotApplEvent()` method returns `SUCCEED` if the connector framework successfully delivers the business object. The poll method checks the return code from `gotApplEvent()` to ensure that the event record's status is updated appropriately. If `gotApplEvent()` returns any return code *except* `FAIL`, the poll method returns `SUCCEED` so that it continues to poll for events. However, on a `FAIL` return code from `gotApplEvent()`, event delivery has failed so the poll method logs an error message and fails.

Table 98 shows the actions that `pollForEvents()` takes based on the `gotApplEvent()` return code.

Table 98. Possible pollForEvents() actions after event delivery with gotApplEvent()

Return code of gotApplEvent()	Actions in pollForEvents()
`SUCCEED`	1. Reset the event status to `SUCCESS`. 2. If the `ArchiveProcessed` connector property is set to true, archive the event and delete it from the event store. 3. Continue polling.
`NO_SUBSCRIPTION_FOUND`	1. Log an error message. 2. Reset the event status to `UNSUBSCRIBED`. 3. If the `ArchiveProcessed` connector property is set to true, archive the event and delete it from the event store. 4. Continue polling.
`CONNECTOR_NOT_ACTIVE`	1. Log an informational message at a trace level of 3. 2. Prepare the event for future re-execution: • For application adapters, reset the event status to `READY_FOR_POLL`. • For technology adapters, push back the event (if possible). 3. Return `SUCCEED` as the `pollForEvents()` outcome status. **Note:** In this case, the event is *not* archived.

Table 98. Possible pollForEvents() actions after event delivery with gotApplEvent() (continued)

Return code of gotApplEvent()	Actions in pollForEvents()
FAIL	1. Log an error message. 2. Reset the event status to ERROR_POSTING_EVENT. 3. If the ArchiveProcessed connector property is set to true, archive the event and delete it from the event store. 4. Return FAIL as the pollForEvents() outcome status.

As Table 98 shows, the action that pollForEvents() takes when the gotApplEvents() method returns an outcome status of CONNECTOR_NOT_ACTIVE depends on the type of connector you have created. For an application connector (in particular a connector whose application uses a database as its event store), the pollForEvents() method should reset the event's status to READY_FOR_POLL to revert an event back to its "unprocessed" state.

However, for technology connectors (in particular, those that do *not* use event tables and therefore cannot always revert an event back to an "unprocessed" state), the connector can hold the event in memory and return an outcome status of SUCCEED from pollForEvents(), rather than attempting to "push" the event back. The connector should keep this event in memory until the adapter is re-activated and pollForEvents() is again invoked. At this time, the connector can try to republish the event.

The following code fragment shows how this functionality might be implemented.

```
BusinessObject eventOnHold;

pollForEvents(...)
{
   ...
   if eventOnHold != null
   {
      event = eventOnHold;
      eventOnHold = null;
   }
   else
   {
      event = getNextUnprocessedEvent();
   }
   ...
   result = gotApplEvent( event );
   if (result == CWConnectorConstant.CONNECTOR_NOT_ACTIVE )
   {
      eventOnHold = event;
      return CWConnectorConstant.SUCCEED;
   }
```

Note: Keep in mind that if you pause the adapter while it is actively processing an event and then later terminate this adapter (or it terminates unexpectedly on its own), "in-doubt" events can result for these events that the connector (using the above logic) has copied to memory. Different adapters have different strategies for how to handle in-doubt events. However, the result of this logic can mean the creation of "in-doubt" events even though the adapter was seemingly terminated properly. These events are *not* lost.

When implementing the pollForEvents() response to the CONNECTOR_NOT_ACTIVE return status, keep in mind that the programming approaches discussed here

assume that the adapter places an event in an ″in-progress″ state while it processes and sends the event to the integration broker. However, not all adapters are implemented this way. An adapter might simply receive an event from a source and then call `gotApplEvent()` to send it to the integration broker. If this adapter terminates in the time between when it receives the event and when it calls `gotApplEvent()`, the event is lost. When such an adapter is restarted, it has no way of reprocessing the event.

Completing the processing of an event

The processing of an event is complete with the completion of the tasks in Table 99.

Table 99. Steps in processing an event

Event-processing task	For more information
The poll method has retrieved the application data for the event and created a business object that represents the event.	“Retrieving application data” on page 187
The poll method has sent the business object to the connector framework.	“Sending the business object to the connector framework” on page 188

Note: For hierarchical business objects, the event processing is complete when the poll method has retrieved the application data for the parent business object and all child business objects and sent the complete hierarchical business object to the connector framework. The event notification mechanism must retrieve and send the entire hierarchical business object, not just the parent business object.

The poll method must ensure that the event status correctly reflects the completion of the event processing. Therefore, it must handle *both* of the following conditions:

- “Handling successful event processing”
- “Handling unsuccessful event processing”

Handling successful event processing: The processing of an event is successful when the tasks in Table 99 successfully complete. The following steps show how the poll method should finish processing a successful event:

1. Receive a “success” return code from the `gotApplEvent()` method signifying the connector framework’s successful delivery of the business object to the messaging system.
2. Copy the event to the archive store. For more information, see “Archiving the event” on page 193.
3. Set the status of the event in the archive store.
4. Delete the event record from the event store.

   Until the event delivery is successful, the poll method should *not* remove the event from the event table.

Note: The order of the steps might be different for different implementations.

Handling unsuccessful event processing: If an error occurs in processing an event, the connector should update the event status to indicate that an error has occurred. Table 100 shows the possible event-status values, based on errors that can occur during event processing.

Table 100. Possible event status after errors in event processing

State of event delivery	Event status	Does polling terminate?
If an error occurs in processing an event	ERROR_PROCESSING_EVENT	No, retrieve the next event from the event store
If the event delivery fails	ERROR_POSTING_EVENT	Yes
If no subscriptions exist for the event	UNSUBSCRIBED	No, retrieve the next event from the event store

For example, if there are no application entities matching the entity key, the event status should be updated to "error processing event". If the event cannot be successfully delivered, its event status should be updated to "error posting event". As discussed in "Sending the business object" on page 189, the poll method should check the return code from `gotApplEvent()` to ensure that any errors that are returned are handled appropriately.

In any case, the event should be left in the event store to be analyzed by a system administrator. When the poll method queries for events, it should exclude events with the error status so that these events are not picked up. Once an event's error condition has been resolved, the system administrator can manually reset the event status so that the event is picked up by the connector on the next poll.

Archiving the event

Archiving an event consists of moving the event record from the event store to an archive store. The Java connector library provides the `CWConnectorEventStore` class to represent an event store, which includes the archive store. Table 101 lists the methods that the Java connector library provides to archive events.

Table 101. Methods for archiving events

Java connector library class	Method
`CWConnectorEventStore`	updateEventStatus(), archiveEvent(), deleteEvent()

Note: For a general introduction to archiving, see "Archiving events" on page 129..

To archive event records from this event store, the poll method takes the following actions:

1. Ensure that archiving is implemented by checking the value of the appropriate connector configuration property, such as `ArchiveProcessed`. For more information, see "Configuring a connector for archiving" on page 130..
2. Copy the event record from the archive store to the event store with the `archiveEvent()` method.

   To provide event archiving, you must implement the `archiveEvent()` method as part of the `CWConnectorEventStore` class. This method identifies the event record to copy by its event ID.

   The `archiveEvents()` method should throw the `ArchiveFailedException` exception if the application is unable to archive the event because it is unable to access the event store. When the `pollForEvents()` method catches this exception, it can return the `APPRESPONSETIMEOUT` outcome status to indicate the lack of response from the application's event store.
3. Update the event status of the archive record with the `updateEventStatus()` method to reflect the reason for archiving the event.

Table 102 shows the likely event-status constants that the archive record will have.

Table 102. Event-status constants in an archive record

Event status	Description
SUCCESS	The event was detected, and the connector created a business object for the event and sent the business object to the connector framework. For more information, see "Handling successful event processing" on page 192.
UNSUBSCRIBED	The event was detected, but there were no subscriptions for the event, so the event was not sent to the connector framework and on to the integration broker. For more information, see "Checking for subscriptions to the event" on page 185.
ERROR_PROCESSING_EVENT	The event was detected, but the connector encountered an error when trying to process the event. The error occurred either in the process of building a business object for the event or in sending the business object to connector framework. For more information, see "Handling unsuccessful event processing" on page 192.

The `updateEventStatus()` method should throw the `StatusChangeFailedException` exception if the application is unable to change the event status because it is unable to access the event store. When the `pollForEvents()` method catches this exception, it can return the `APPRESPONSETIMEOUT` outcome status to indicate the lack of response from the application's event store.

4. Delete the event record from the event store with the `deleteEvent()` method.

   You must implement the `deleteEvent()` method as part of the `CWConnectorEventStore` class. This method uses the event ID to identify the event record to delete.

   The `deleteEvents()` method should throw the `DeleteFailedException` exception if the application is unable to delete the event because it is unable to access the event store. When the `pollForEvents()` method catches this exception, it can return the `APPRESPONSETIMEOUT` outcome status to indicate the lack of response from the application's event store.

Figure 65 contains a code fragment that archives an event.

```
// Archive the event if ArchiveProcessed is set to true.
if (arcProcessed.equalsIgnoreCase("true")) {
   // Archive the event in the application's archive store.
   evts.archiveEvent(evtObj.getEventID());

   // Delete the event from the event store.
   evts.deleteEvent(evtObj.getEventID());
   }
```

Figure 65. Archiving an event

After archiving is complete, your poll method should set the appropriate return code:

- If the archiving takes place after an event is successfully delivered, the return code is "success", indicated with the `SUCCEED` outcome-status constant.
- If archiving is due to some error condition (such as unsubscribed events or an error in processing the event), the poll method might need to return a "fail" status, indicated with the `FAIL` outcome-status constant.

Releasing event-store resources

Often, the `pollForEvents()` method needs to allocate resources to access the event store. To prevent excessive memory usage by these resources, you can release them at the end of the poll method. Table 103 lists the methods that the Java connector library provides to release event-store resources.

Table 103. Method for releasing event-store resources

Java connector library class	Method
`CWConnectorEventStore`	cleanupResources()

For example, if the event store is implemented as event tables in a database, `pollForEvents()` might allocate SQL cursors to access these tables. You can implement a `cleanupResources()` method to free these SQL cursors. At the end of `pollForEvents()`, you can then call `cleanupResources()` to free the memory that these cursors use.

Note: The `CWConnectorEventStore` class does *not* provide a default implementation of the `cleanupResources()` method. To free event-store resources, you must override `cleanupResources()` with a version that releases the resources needed to access your event store.

Default implementation of the Java pollForEvents()

Figure 66 shows the default implementation of the `pollForEvents()` in the `CWConnectorAgent` class. You can use this default implementation, which follows the basic logic outlined in "Basic logic for pollForEvents()" on page 129, or you can override this method with your own implementation.

```
/**
 * Default implementation of pollForEvents.
 */
public int pollForEvents() {
   CWConnectorUtil.traceWrite(
      CWConnectorLogAndTrace.LEVEL5,"Entering pollForEvents.");

   // Get the EventStoreFactory implementation name from the
   // getEventStore() method.
   CWConnectorEventStore evts=getEventStore();
   if (evts==null)
   {
      CWConnectorUtil.generateAndLogMsg(10533,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 0);
      return CWConnectorConstant.APPRESPONSETIMEOUT
   }
   try { //finally block
       // Fetch PollQuantity number of events from the application.
      try {
         evts.fetchEvents();
      } catch (StatusChangeFailedException e) {
         CWConnectorUtil.generateAndLogMsg(10533,
            CWConnectorLogAndTrace.XRD_ERROR,0,0);
         CWConnectorUtil.logMsg(e.getMessage());
         e.printStackTrace();
         return CWConnectorConstant.APPRESPONSETIMEOUT;
      }

   // Get the property values for PollQuantity and ArchiveProcessed.
   int pollQuantity;
   String poll=CWConnectorUtil.getConfigProp("PollQuantity");
   try {
      if (poll == null || poll.equals(""))
         pollQuantity=1;
      else
         pollQuantity=Integer.parseInt(poll);
   } catch (NumberFormatException e) {
      CWConnectorUtil.generateAndLogMsg(10544,
         CWConnectorLogAndTrace.XRD_ERROR, 0);
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.FAIL;
   }

   String arcProcessed=CWConnectorUtil.getConfigProp(
      "ArchiveProcessed");

   // In case the ArchiveProcessed property is not set, use true
   // as default.
   if (arcProcessed == null || arcProcessed.equals(""))
      arcProcessed=CWConnectorAttrType.TRUESTRING;
   CWConnectorEvent evtObj;
   CWConnectorBusObj bo=null;
```

Figure 66. Implementation of basic logic for pollForEvents() (Part 1 of 7)

```
try {
   for (int i=0; i < pollQuantity; i++){

      // Process each event retrieved from the application.
      // Get the next event to be processed.
      evtObj=evts.getNextEvent();

      // A null return indicates that there were no events with
      // READY_FOR_POLL status. Return SUCCESS.
      if (evtObj == null) {
         CWConnectorUtil.generateAndLogMsg(10534,
            CWConnectorLogAndTrace.XRD_INFO,0,0);
         return CWConnectorConstant.SUCCEED;
      }
      // Check if the connector has subscribed to the event
      // generated for the business object.
      boolean isSub=isSubscribed(evtObj.getBusObjName(),
         evtObj.getVerb());
      if (isSub) {
         // Retrieve the complete CWConnectorBusObj corresponding
         // to the object using the getBO method in
         // CWConnectorEventStore. This method sets the verb on a
         // temporary business object to RetrieveByContent
         // and retrieves the corresponding data information to be
         // filled in the business object from the application.
         try {
            bo = evts.getBO(evtObj);
            // Terminate flag will be set in the event store when
            // the doVerbFor method returns APPRESPONSETIMEOUT in
            // getBO.
            if (evts.getTerminate())
               return CWConnectorConstant.APPRESPONSETIMEOUT;
         }catch (AttributeNotFoundException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","AttributeNotFoundException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
         }catch (SpecNameNotFoundException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","SpecNameNotFoundException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
```

Figure 66. Implementation of basic logic for pollForEvents() (Part 2 of 7)

```
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
         }catch (InvalidVerbException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","InvalidVerbException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
         }catch (WrongAttributeException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","WrongAttributeException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
         }catch (AttributeValueException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","AttributeValueException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
```

Figure 66. Implementation of basic logic for pollForEvents() (Part 3 of 7)

```
      }catch (AttributeNullValueException e) {
         CWConnectorUtil.generateAndLogMsg(10536,
            CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
            "getBO","AttributeNullValueException");
         CWConnectorUtil.logMsg(e.getMessage());
         e.printStackTrace();
         // Update the event status to ERROR_PROCESSING_EVENT
         evts.updateEventStatus(evtObj,
            CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
         if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
         {
            // Archive the event in the application's archive store
            evts.archiveEvent(evtObj.getEventID());
            // Delete the event from the event store
            evts.deleteEvent(evtObj.getEventID());
         }
         continue;
      }

      // Log a fatal error in case the object is not found.
      if (evtObj.getStatus()==
   CWConnectorEventStatusConstants.ERROR_OBJECT_NOT_FOUND) {
         CWConnectorUtil.generateAndLogMsg(10543,
            CWConnectorLogAndTrace.XRD_FATAL,0,0);
         // Update the event status to ERROR_OBJECT_NOT_FOUND
         evts.updateEventStatus(evtObj,
            CWConnectorEventStatusConstants.ERROR_OBJECT_NOT_FOUND);
         if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
         {
            // Archive the event in the application's archive store
            evts.archiveEvent(evtObj.getEventID());
            // Delete the event from the event store
            evts.deleteEvent(evtObj.getEventID());
         }
         continue;
      }
      // In case the business object is null, the retrieve call
      // returned an error.
      if (bo == null) {
         CWConnectorUtil.generateAndLogMsg(10335,
            CWConnectorLogAndTrace.XRD_ERROR,0,0);
         // Update the event status to ERROR_PROCESSING_EVENT
         evts.updateEventStatus(evtObj,
            CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
         if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
         {
            // Archive the event in the application's archive store
            evts.archiveEvent(evtObj.getEventID());
            // Delete the event from the event store
            evts.deleteEvent(evtObj.getEventID());
         }
         continue;
      }
```

Figure 66. Implementation of basic logic for pollForEvents() (Part 4 of 7)

```
// Set the processing verb on the business object.
try {
   bo.setVerb(evtObj.getVerb());
} catch(InvalidVerbException e){
   CWConnectorUtil.generateAndLogMsg(10536,
      CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
      "setVerb","InvalidVerbException");
   CWConnectorUtil.logMsg(e.getMessage());
   e.printStackTrace();
   // Update the event status to ERROR_PROCESSING_EVENT
   evts.updateEventStatus(evtObj,
      CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
   if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
   {
      // Archive the event in the application's archive store
      evts.archiveEvent(evtObj.getEventID());
      // Delete the event from the event store
      evts.deleteEvent(evtObj.getEventID());
   }
   continue;
}
// Check again for subscription.
if (isSubscribed(bo.getName(),bo.getVerb())){
   // Send the event to integration broker.
   int stat=gotApplEvent(bo);
   if (stat == CWConnectorConstant.CONNECTOR_NOT_ACTIVE){
      CWConnectorUtil.generateAndTraceMsg(
         CWConnectorLogAndTrace.LEVEL3, 10551,
         CWConnectorLogAndTrace.XRD_INFO, 0, 0);
      evts.updateEventStatus(evtObj,
         CWConnectorEventStatusConstants.READY_FOR_ROLL);
      // No need to archive the event, as the status is reset to
      // READY_FOR_POLL. It is as if this event never reached the
      // connector for processing.
      return CWConnectorConstant.SUCCEED;
   }
   if (stat == CWConnectorConstant.NO_SUBSCRIPTION_FOUND){
      CWConnectorUtil.generateAndLogMsg(10552,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 0);
      // Update the event status to UNSUBSCRIBED.
      evts.updateEventStatus(evtObj,
         CWConnectorEventStatusConstants.UNSUBSCRIBED);
      if (arcProcessed.equalsIgnoreCase(
            CWConnectorAttrType.TRUESTRING)) {
         // Archive the event in the application's archive store
         evts.archiveEvent(evtObj.getEventID());
         // Delete the event from the event store
         evts.deleteEvent(evtObj.getEventID());
      }
      continue;
   }
   if (stat == CWConnectorConstant.SUCCEED){
      // Update the event status to SUCCESS.
      evts.updateEventStatus(evtObj,
         CWConnectorEventStatusConstants.SUCCESS);
      if (arcProcessed.equalsIgnoreCase(
            CWConnectorAttrType.TRUESTRING)) {
         // Archive the event in the application's archive store
         evts.archiveEvent(evtObj.getEventID());
         // Delete the event from the event store
         evts.deleteEvent(evtObj.getEventID());
      }
      continue;
```

Figure 66. Implementation of basic logic for pollForEvents() (Part 5 of 7)

```
                } else // gotApplEvent returned FAIL
                {
                   CWConnectorUtil.generateAndLogMsg(10532,
                      CWConnectorLogAndTrace.XRD_ERROR,0,0);
                   // Update the event status to ERROR_POSTING_EVENT.
                   evts.updateEventStatus(evtObj,
             CWConnectorEventStatusConstants.ERROR_POSTING_EVENT);
                   // Archive the event if ArchiveProcessed is set
                   // to true.
                   if (arcProcessed.equalsIgnoreCase(
                         CWConnectorAttrType.TRUESTRING)) {
                      // Archive the event in the application's
                      // archive store.
                      evts.archiveEvent(evtObj.getEventID());
                      // Delete the event from the event store.
                      evts.deleteEvent(evtObj.getEventID());
                   }
                   return CWConnectorConstant.FAIL;
                }

                }  else // Event unsubscribed.
                {
                   CWConnectorUtil.generateAndLogMsg(10552,
                      CWConnectorLogAndTrace.XRD_ERROR, 0, 0);
                   // Update the event status to UNSUBSCRIBED.
                   evts.updateEventStatus(evtObj,
                      CWConnectorEventStatusConstants.UNSUBSCRIBED);
                   // Archive the event if ArchiveProcessed is set
                   // to true.
                   if (arcProcessed.equalsIgnoreCase(
                         CWConnectorAttrType.TRUESTRING)) {
                      // Archive the event in the application's
                      // archive store.
                      evts.archiveEvent(evtObj.getEventID());
                      // Delete the event from the event store.
                      evts.deleteEvent(evtObj.getEventID());
                   }
                   continue;
                }

             } else
             {
                CWConnectorUtil.generateAndLogMsg(10552,
                   CWConnectorLogAndTrace.XRD_ERROR, 0, 0);
                // Update the event status to UNSUBSCRIBED.
                evts.updateEventStatus(evtObj,
                   CWConnectorEventStatusConstants.UNSUBSCRIBED);
                // Archive the event if ArchiveProcessed is set
                // to true.
                if (arcProcessed.equalsIgnoreCase(
                      CWConnectorAttrType.TRUESTRING)) {
                   // Archive the event in the application's
                   // archive store.
                   evts.archiveEvent(evtObj.getEventID());
                   // Delete the event from the event store.
                   evts.deleteEvent(evtObj.getEventID());
                }
                continue;
             }
          } //For loop
}
```

Figure 66. Implementation of basic logic for pollForEvents() (Part 6 of 7)

```
   } catch (StatusChangeFailedException e){
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "updateEventStatus","StatusChangeFailedException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.APPRESPONSETIMEOUT;
   } catch (InvalidStatusChangeException e){
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "updateEventStatus","InvalidStatusChangeException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.APPRESPONSETIMEOUT;
   } catch (ArchiveFailedException e){
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "archiveEvent","ArchiveFailedException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.APPRESPONSETIMEOUT;
   } catch (DeleteFailedException e){
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "deleteEvent","DeleteFailedException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.APPRESPONSETIMEOUT;
   } catch (AttributeNullValueException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "get method in event store","AttributeNullValueException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.FAIL;
   }
   } finally {
      evts.cleanupResources();
   }
   return CWConnectorConstant.SUCCEED;
}
```

Figure 66. Implementation of basic logic for pollForEvents() (Part 7 of 7)

Shutting down the connector

In the Java connector library, the `terminate()` method for a Java connector is defined in the `CWConnectorAgent` class. Typical return codes used in `terminate()` are `SUCCEED` and `FAIL`. Figure 67 shows a sample `terminate()` method for a Java connector.

```
public int terminate(){

   CWConnectorUtil.traceWrite(CWConnectorLogAndTrace.LEVEL4,
      "Entering Connector terminate()");

   //  disconnect from application
    boolean logoutSuccessful = userConnect.logout();

   // free any resources, logoff any cache sessions if connection
   // pool is used.

   CWConnectorUtil.traceWrite(CWConnectorLogAndTrace.LEVEL4,
   return CWConnectorConstant.SUCCEED;
}
```

Figure 67. Java terminate() method

Handling errors and status

This section provides the following information about how the methods of the connector class library indicate error conditions:

- "Java return codes"
- "Exceptions" on page 204
- "Return-status descriptor" on page 206

Note: You can also use error logging and message logging to handle error conditions and messages in your connector. For more information, see Chapter 6, "Message logging," on page 139

Java return codes

In the Java connector library, the outcome-status constants in the `CWConnectorConstant` class define the Java return codes. Table 104 lists these Java outcome-status constants.

Table 104. Java outcome-status codes

Return code	Description
`CWConnectorConstant.SUCCEED`	The operation succeeded.
`CWConnectorConstant.FAIL`	The operation failed.
`CWConnectorConstant.APPRESPONSETIMEOUT`	The application is not responding.
`CWConnectorConstant.MULTIPLE_HITS`	The connector found multiple matching records when retrieving using non-key values. The first record is returned with this status code.
`CWConnectorConstant.BO_DOES_NOT_EXIST`	The connector performed a Retrieve operation, but the entity that the business object represents does not exist in the application database.
`CWConnectorConstant.RETRIEVEBYCONTENT_FAILED`	The connector was not able to find matches for retrieve by non-key values.
`CWConnectorConstant.UNABLETOLOGIN`	The connector is unable to log in to the application.
`CWConnectorConstant.VALCHANGE`	At least one value in a business object has changed.
`CWConnectorConstant.VALDUPES`	The object in the application already has the requested data values.
`CWConnectorConstant.CONNECTOR_NOT_ACTIVE`	The connector is not active; it has been paused.
`CWConnectorConstant.NO_SUBSCRIPTION_FOUND`	No subscriptions were found for the event.

Outcome-status constants are provided for use in user implementations of many of the Java methods, as Table 105 shows. Although your code can return these values from within any method, some of the return codes were designed with specific uses in mind. For example, VALCHANGE informs the integration broker that the connector is sending a business object with changed values.

Table 105. Outcome-status values for Java connector methods

Connector method	Possible outcome-status codes
archiveEvent()	SUCCEED, FAIL
doVerbFor()	SUCCEED, FAIL, APPRESPONSETIMEOUT, VALCHANGE, VALDUPES, MULTIPLE_HITS, RETRIEVEBYCONTENT_FAILED, BO_DOES_NOT_EXIST
gotApplEvent()	SUCCEED, FAIL, CONNECTOR_NOT_ACTIVE, NO_SUBSCRIPTION_FOUND
pollForEvents()	SUCCEED, FAIL, APPRESPONSETIMEOUT
terminate()	SUCCEED, FAIL

The outcome-status constant that the connector framework receives helps to determine its next action, as follows:

- If the outcome status is APPRESPONSETIMEOUT, the connector framework shuts down the connector.

  When the connector framework receives this outcome status, it copies the APPRESPONSETIMEOUT status into the return-status descriptor and returns this descriptor to inform the connector controller that the application is *not* responding. Once it has sent this return-status descriptor, the connector framework stops the process in which the connector runs. A system administrator must fix the problem with the application and restart the connector to continue processing events and business object requests.
- For *all* other outcome-status values, the connector framework continues execution of the connector.

  During request processing, the connector framework copies the outcome status into the status field of the return-status descriptor and includes this descriptor in its response to the integration broker. It continues execution of the connector. For some outcome-status values, the connector framework also includes a response business object in its response. For more information, see "Updating the request business object" on page 170.

Important: The connector framework does *not* stop execution of the connector when it receives the FAIL outcome-status constant.

Exceptions

In addition to returning status codes, the methods of the Java connector library can throw exceptions to indicate certain predefined conditions. This section provides the following information about how to handle exceptions in a Java connector:

- "What Is a Java connector exception?"
- "Exceptions from the Java connector library" on page 205

What Is a Java connector exception?

When a method of the Java connector library throws an exception, this exception object is a subclass of the CWException class, which is an extension of the Java Exception class. As Figure 68 shows, this *exception object* contains a message and

status, as well as an exception-detail object with additional information about the exception.

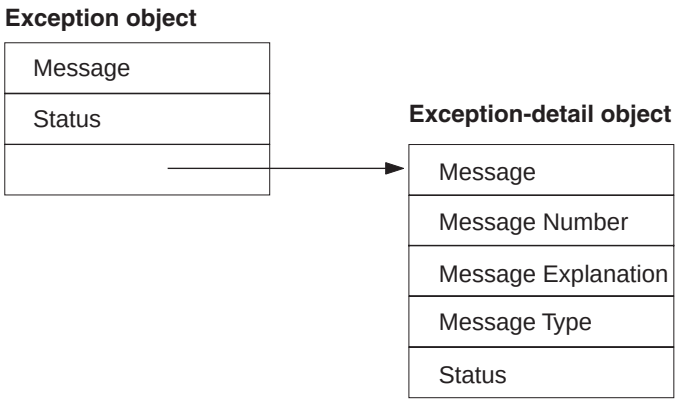

Figure 68. The CWException exception object

Table 106 shows the accessor methods that the `CWException` class provides to obtain information in the exception object.

Table 106. Information in the exception object

Member	Accessor method
Message text	getMessage()
Status	getStatus(), setStatus()
Exception-detail object	getExceptionObject()

Note: For more information on the methods in the `CWException` class, see Chapter 24, "CWException class," on page 383.

The *exception-detail object* is an instance of the `CWConnectorExceptionObject` class. As Figure 68 shows, an exception object contains an exception-detail object. This exception-detail object provides more detailed information about the Java connector library exception, as Table 107 shows.

Table 107. Information in the exception-detail object

Member	Description	Accessor method
Message text	The message text for the exception	getMsg(), setMsg()
Message number	The number in a message file that identifies the message	getMsgNumber(), setMsgNumber()
Message explanation	The detailed description of a message, which is also stored in the message file. This information might include a corrective action.	getExpl(), setExpl()
Message type	An integer constant that indicates the severity of a message	getMsgType(), setMsgType()
Status	An integer status that indicates the outcome of the method.	getStatus(), setStatus()

Note: For more information on the methods in the `CWConnectorExceptionObject` class, see Chapter 19, "CWConnectorExceptionObject class," on page 335.

Exceptions from the Java connector library

When you write code for a Java connector, you can include Java `try` and `catch` statements to handle specific exceptions thrown by the methods of the Java

connector library. The reference description for most Java connector library methods has a section entitled Exceptions, which lists the exceptions thrown by that method.

Figure 69 shows a code fragment from the default implementation of the `pollForEvents()` method that catches the exceptions that the `getBO()` method throws.

```
try {
   bo = evts.getBO(evtObj);

   }catch (AttributeNotFoundException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","AttributeNotFoundException");
      return CWConnectorConstant.FAIL;
   }catch (SpecNameNotFoundException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","SpecNameNotFoundException");
      return CWConnectorConstant.FAIL;
   }catch (InvalidVerbException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
            0,2,"getBO","InvalidVerbException");
      return CWConnectorConstant.FAIL;
   }catch (WrongAttributeException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","WrongAttributeException");
      return CWConnectorConstant.FAIL;
   }catch (AttributeValueException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","AttributeValueException");
      return CWConnectorConstant.FAIL;
   }catch (AttributeNullValueException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","AttributeNullValueException");
      return CWConnectorConstant.FAIL;
   }
```

Figure 69. Catching exceptions from getBO()

When a Java connector library method throws an exception, it does not usually provide message and status information in the exception object. However, you can choose to fill the exception object with additional information as needed.

Return-status descriptor

The *return-status descriptor* usually contains information about the success (or lack thereof) of the verb processing that the business object handler (the `doVerbFor()` method) has performed. The calling code can use this status information to determine how to proceed. When the business object handler for a particular business object is invoked, the `doVerbFor()` of its associated business-object-handler class executes. However, the actual method invoked is not the user-implemented `doVerbFor()` (which the connector developer implements as part of the business-object-handler class). Instead, the business object handler invokes a low-level `doVerbFor()` method, which is defined in this same class but which the connector developer does not implement.

This low-level doVerbFor() method performs the following tasks:

1. Receive an empty return-status descriptor as an argument.
2. Call the user-implemented doVerbFor() to perform the verb processing.
3. Populate the return-status descriptor based on the verb-processing status when this user-implemented doVerbFor() completes (either successfully or otherwise).

Because the low-level doVerbFor() receives an instantiated return-status descriptor as an argument, any changes that it makes to this return-status descriptor are available to the calling code (which instantiated the return-status descriptor) once the low-level doVerbFor() exits. Therefore, the code that called the business object handler can access this return-status descriptor to obtain information about the status of the verb processing.

Access to this return-status descriptor can be performed in either of the following ways:

- "Implicitly accessing the return-status descriptor"
- "Explicitly accessing the return-status descriptor"

Implicitly accessing the return-status descriptor

In request processing, the connector framework uses the return-status descriptor to report the status of the verb processing back to the integration broker. When the connector framework receives a request business object, it locates the associated business-object-handler class and invokes its low-level doVerbFor() method. It passes to this low-level doVerbFor() an instantiated, empty return-status descriptor.

When the low-level doVerbFor() completes, it has populated the return-status descriptor with the verb-processing status from the user-implemented doVerbFor() method. The connector framework then includes this return-status descriptor as part of its response to the integration broker. For more information, see "Populating the return-status descriptor" on page 170.

Explicitly accessing the return-status descriptor

In event notification, the poll method can use the return-status descriptor to determine the success of the retrieval of application data associated with an event. When the poll method, pollForEvents(), retrieves an event from the event store, the event usually contains only the key values of the associated application event. To obtain all application data, pollForEvents() must use the key value (or values) to query the application and retrieve the full set of values. For more information, see "Retrieving application data" on page 187.

A common way to retrieve this application data is to call the business object handler with a RetrieveByContent verb in the business object. To facilitate this use of a business object handler, the CWConnectorBusObj class provides a version of the doVerbFor() method. When calling code calls this doVerbFor() method, it invokes the business object handler for the current business object by calling the low-level doVerbFor() method. The code that calls the CWConnectorBusObj version of doVerbFor() must first create a return-status descriptor and then pass this instantiated, empty return-status descriptor into doVerbFor().

The CWConnectorBusObj version of doVerbFor() passes the empty return-status descriptor to the low-level doVerbFor() method in the business-object-handler class. When the low-level doVerbFor() completes, it has populated the return-status descriptor with the verb-processing status from the user-implemented doVerbFor() method. The CWConnectorBusObj version of doVerbFor() passes this return-status

descriptor back to the calling code. Because the calling code has instantiated this return-status descriptor, it can explicitly access its contents to determine the success of the verb processing.

For a Java connector, the return-status descriptor is a `CWConnectorReturnStatusDescriptor` object. Table 108 lists the status information that this structure provides.

Table 108. Information in the return-status descriptor

Return-status descriptor information	Description	Java accessor method
Error message	A string to provide a description of the error condition	getErrorString(), setErrorString()
Status	An additional status value to further detail the cause of the error condition	getStatus(), setStatus()

The `CWConnectorEventStore` class provides the `getBO()` method to retrieve application data associated with an event. The default implementation of the `getBO()` method calls the `CWConnectorBusObj` version of `doVerbFor()` to perform this retrieval. The default implementation of the `pollForEvents()` method includes a call to `getBO()`. Therefore, your `pollForEvents()` does not need to explicitly access the return-status descriptor for information about the retrieval status in either of the following cases:

- If you use the default implementation of `pollForEvents()`
- If you call the default implementation of `getBO()` in your own `pollForEvents()` method

The default implementation of `getBO()` automatically accesses the return-status descriptor and returns values (or throws exceptions) to indicate the retrieval status.

Note: You can use the methods of the `CWConnectorReturnStatusDescriptor` method to access the collaboration status from a return-status descriptor after execution of the `executeCollaboration()` method.

Important: Any status code that the `doVerbFor()` method sets in the return-status descriptor must have meaning to the collaboration. The collaboration developer and the connector developer must agree on the meaning of this status code.

Chapter 8. Adding a connector to the business integration system

To run in the IBM WebSphere business integration system, a connector must be defined in the repository. Pre-defined adapters, which the WebSphere Business Integration Adapters product provides, have predefined connector definitions in the repository. A system administrator need only configure the application and set the connector's configuration properties to run the connector.

For the IBM WebSphere business integration system to be able to access a connector that you have developed, you must take the following steps:

1. Create the connector definition in the repository.
2. If WebSphere MQ will be used for messaging between connector components, add message queues for the connector.
3. Create the connector's initial configuration file.
4. Create the connector's startup script.

This chapter provides information on adding a new connector to the IBM WebSphere business integration system. This chapter includes the following sections:

Naming the connector

This chapter provides suggested naming conventions for the files and directories used in connector development. Naming conventions provide a way to make you connector files more easy to locate and identify. Table 109 summarizes the suggested naming conventions for connector files. Many of these files are based on the *connector name*, which should uniquely identify it within the WebSphere business integration system. This name (*connName*) can identify the application or technology with which the connector communicates.

Table 109. Suggested naming conventions for a connector

Connector file	**Name**
Connector definition	*connName*Connector
Connector directory	*ProductDir*\connectors*connName*
Initial connector configuration file	File name: BIA_CN_*connName*.txt Directory name: *ProductDir*\repository*connName*
User-customized connector configuration file	File name: CN_*connName*.txt Directory name: *ProductDir*\connectors*connName*
Connector class	*connName*Agent.java

Table 109. Suggested naming conventions for a connector (continued)

Connector file	Name
Connector library	Java jar file: *connDir*\BIA_*connName*.jar Java package: com.crossworlds.connectors.*connName*. where *connDir* is the name of the connector directory, as defined above.
Connector startup script	Windows platforms: *connDir*\start_*connName*.bat UNIX-based platforms: *connDir*\connector_manager_*connName*.sh where *connDir* is the name of the connector directory, as defined above.

For more information on naming conventions for connectors, see *Naming IBM WebSphere InterChange Server Components* in the IBM WebSphere InterChange Server documentation set.

Compiling the connector

Once you have written the connector's application-specific component, you must compile it into an executable format, its *connector library*. This section provides information on how to compile a connector.

To compile a Java connector, take the following steps:

- Use a JDK 1.3.1 development environment. For more information, see "Setting up the development environment" on page 30.
- Ensure that both of the following files are in the `lib` subdirectory of the product directory.
  - `crossworlds.jar`
  - `WBIA.jar`
- Include `crossworlds.jar` in the project path. Also include in the project path any application-specific jar files that your connector's application-specific component requires.
- Compile the connector source (.java) files into class (.class) files with the Java compiler.
- Create the Java connector's library file, which is a Java archive (jar) file that contains the compiled Java code.

  The suggested naming convention for the jar file is to begin its name with the string `"BIA_"`. Follow this string with the connector name, which uniquely identifies the connector (see Table 109 on page 209). For more information about the connector name, see "Naming the connector" on page 209.

  For example, for a Java connector with a connector name of `MyJava`, you could name its jar file as:

  ```
  BIA_MyJava.jar
  ```

Creating the connector definition

To run in the IBM WebSphere business integration system, a connector must be defined in the *repository*. Pre-defined adapters, which the WebSphere Business Integration Adapters product provides, have predefined connector definitions that are loaded in the repository at installation time. To run a predefined connector, a system administrator need only configure the application and set the connector's configuration properties. However, before the IBM WebSphere business integration system can access a connector that you have developed, you must take the following steps:

- Create a connector definition to define the connector within the repository.
- Create an initial configuration file to assist users in connector configuration (optional).

Defining the connector

To define the connector within the WebSphere business integration system, you create a *connector definition*. This connector definition includes the following information to define the connector in the repository:

- The name of the connector definition
- Supported business objects and associated maps
- Connector configuration properties

A tool called Connector Configurator collects this information and stores it in the repository.

WebSphere InterChange Server

When your integration broker is InterChange Server, the repository is a database that InterChange Server communicates with to obtain information about components in the WebSphere business integration system. In this repository, connector definitions reside. These connector definitions include both standard and connector-specific connector configuration properties that the connector controller and the client connector framework require. The connector can also have a local configuration file, which provides configuration information for the connector locally. When a local configuration file exists, it takes precedence over the information in the InterChange Server repository.

You update the connector definitions in the InterChange Server repository with Connector Configurator from within the System Manager tool. You can update the locale configuration file with the standalone version of Connector Configurator, which resides in the `bin` subdirectory of your product directory.

WebSphere MQ Integrator Broker

When your integration broker is WebSphere MQ Integrator Broker, the repository is a directory of files that the connector framework uses to obtain information about components of the WebSphere business integration system. In this repository, connector definitions for each adapter in the system resides.

You update the connector definitions in the local repository with Connector Configurator, which resides in the `bin` subdirectory of your product directory.

For information on how to use Connector Configurator, refer to Appendix B, "Connector Configurator," on page 495.

The connector definition name

The connector definition name uniquely identifies the connector within the WebSphere business integration system. By convention, a connector definition name usually takes the following form:

*connName*Connector

where *connName* is the connector name (see Table 109 on page 209). For more information on the connector name, see "Naming the connector" on page 209. For example, if the connector name is MyConn, the name of its connector definition is MyConnConnector.

Supported business objects and maps

A connector definition must specify the following information about the business objects that the connector supports:

- The business object definitions

  Each business object that the connector is able to send to or receive from the integration broker must be specified as a supported business object. Connector Configurator provides a Supported Business Objects tab in which you can enter the connector's supported business objects.

  Note: All application-specific business objects that the connector supports must be defined in the repository *before* you can include them as supported business objects in the connector definition. For information on how to define application-specific business objects, see the *Business Object Development Guide*.

- Associated maps

WebSphere InterChange Server

Only the connector definition for a connector that communicates with InterChange Server as its integration broker includes the maps associated with the connector. Associated maps are those maps that convert between the connector's application-specific business objects and the appropriate generic business objects.

Connector Configurator provides an Associated Maps tab in which you can enter the connector's associated maps.

Connector configuration properties

The connector definition also contains the connector configuration properties. To initialize these properties, you must take the following steps:

- Assign values for standard connector configuration properties.
- Define any connector-specific configuration properties that your connector uses and assign them values as appropriate.

Connector Configurator provides two tabs for specifying connector configuration properties: Standard Properties and Connector-Specific Properties. For more information on connector configuration properties, see "Using connector configuration property values" on page 70..

Creating the initial configuration file

By convention, pre-defined adapters provide an initial configuration file for users to use the first time that they configure the adapter with Connector Configurator. The suggested name for this configuration file is:

`BIA_CN_connName.txt`

where *connName* is the connector name (see Table 109 on page 209). For more information on the connector name, see "Naming the connector" on page 209. This initial configuration file resides in the following directory:

`ProductDir\repository\connName`

That is, the `repository` subdirectory of the product directory contains directories for each connector. Each connector's directory (*connName*) is named with its unique connector name and within this directory resides the initial configuration file with the following name.

For users to configure a connector that you have developed, you can provide an initial configuration file for your new connector. As part of your connector development, you have probably specified the settings for the standard configuration properties as well as defining any connector-specific configuration properties. This connector configuration information should reside in your repository. However, once your connector is moved to some other environment, it loses access to this repository. Therefore, you should create an initial configuration file that is part of your released connector.

To create this initial configuration file, bring up Connector Configurator for your connector and save its configuration in the following file:

`ProductDir\repository\connName\BIA_CN_connName.txt`

Note: These steps assume that during the course of development, you have already created a connector configuration file (.cfg) for your connector. The preceding step just saves this connector configuration information in a separate file, which is included as part of the released connector.

Starting up a new connector

To start up the connector, you execute a *connector startup script*. As Table 110 shows, the name of this startup script depends on the operating system which you are using.

Table 110. Startup scripts for a connector

Operating system	Startup script
UNIX-based systems	`connector_manager_connName`
Windows	`start_connName.bat`

The startup script supports those adapters that the WebSphere Business Integration Adapters product provides. To start up a predefined connector, a system administrator runs its startup script. The startup scripts for most predefined connectors expect the following command-line arguments:

1. The first argument is the connector name, which identifies the following:
   - The name of the connector's directory under the `connectors` subdirectory of the product directory

- The connector library, which resides in the connector's directory

2. The second argument is the name of the integration broker instance against which the connector runs.

WebSphere InterChange Server

When your integration broker is InterChange Server (ICS), the startup script specifies the name of the ICS instance against which your connector runs. On Windows systems, this ICS instance name (which was specified in the installation process) appears in each of the connector shortcuts of the startup script.

Other integrator brokers

When your integration broker is a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the startup script specifies the name of the broker instance against which your connector runs. On Windows systems, this instance name (which was specified in the installation process) appears in each of the connector shortcuts of the startup script.

3. Optional additional startup parameters can be specified on the command line and are passed to the connector runtime.

   For more information about the startup parameters, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set or your implementation guide in the WebSphere Business Integration Adapters documentation set.

WebSphere InterChange Server

Before you start a connector, InterChange Server must be running for the connector to complete its initialization and obtain its business objects from the repository.

Before you can start up a connector that you have developed, you need to ensure that a startup script supports your new connector. To enable a startup script to start your own connector, you must take the following steps:

1. Prepare a connector directory for your connector.
2. Create the startup script for your connector. For Windows systems, also create a shortcut for your connector startup.
3. Set up the startup script as a Windows service (optional).

The following sections describe each of these steps.

Preparing the connector directory

The *connector directory* contains the runtime files for your connector. To prepare the connector directory, take the following steps:

1. Create a connector directory for your new connector under the `connectors` subdirectory of the product directory:

   ProductDir`\connectors\`*connName*

By convention, this directory name matches the connector name (*connName*). The connector name is a string that uniquely identifies the connector. For more information, see "Naming the connector" on page 209.

2. Move your connector's library file to this connector directory.

   A Java connector's library file is a Java archive (jar) file. You created this jar file when you compiled the connector. For more information, see "Compiling the connector" on page 210.

Creating startup scripts

As Table 110 on page 213 shows, a connector requires a startup script for the system administrator to start execution of the connector process. The startup script to use depends on the operating system on which you are developing your connector.

Startup script and shortcut on Windows systems

Starting a connector on a Windows system involves the following steps:

1. Call the connector's startup script, `start_connName.bat`.

   The `start_connName.bat` script (where *connName* is the name of your connector) is a connector-specific startup script. It provides connector-specific information (such as application-specific libraries and their locations). By convention, this script resides in the connector directory:

   `ProductDir\connectors\connName`

   It is this `start_connName.bat` script that the user invokes to start the connector on a Windows system.

2. Call the generic connector-invocation script, `start_adapter.bat`

   The `start_adapter.bat` file is generic to all connectors. It performs the actual invocation of the connector within the JVM. It resides in the `bin` subdirectory of the product directory. The `start_connName.bat` script must call the `start_adapter.bat` script to actually invoke the connector.

Figure 70 shows the steps to start a connector on a Windows system.

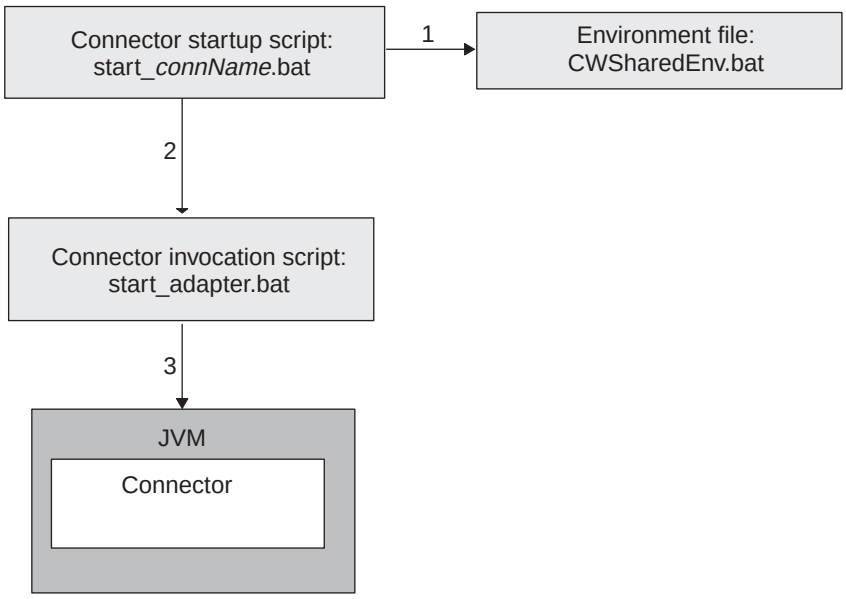

Figure 70. Starting a connector on a Windows system

When a WebSphere Business Integration Adapters Installer installs a predefined connector on a Windows system, it takes the following steps:

- Install a startup script for the predefined connector.
- Create a menu option for the predefined connector under the `Programs > IBM WebSphere Business Integration Adapters > Adapters > Connectors` menu.

To provide the ability to start up your own connector, you must duplicate these steps by:

- Generating the `start_`*`connName`*`.bat` startup script and putting it in the `connector\`*`connName`* subdirectory of the product directory
- Providing a menu option for the connector under the `Programs > IBM WebSphere Business Integration Adapters > Adapters > Connectors` menu. Each menu option is a shortcut that invokes the Windows startup script, `start_`*`connName`*`.bat`, for the particular connector.

Creating the startup script: To create a custom connector startup script, you create a new connector-specific startup script called `start_`*`connName`*`.bat` (where *connName* is your Java connector name). For example, if your Java connector has a connector name of `MyJava`, its startup script name is `start_MyJava.bat`. As a starting point, you can copy the startup-script template, which is located in the following file:

`ProductDir``\templates\start_connName.bat`

Figure 71. shows a sample of the contents of the startup-script template for Windows. Please consult the version of this file released with your product for the most current contents.

```
REM A sample of start_connName.bat which calls start_adapter.bat
@echo off

call "%ADAPTER_RUNTIME%"\bin\wbia_connEnv.bat
setlocal

REM If required, goto the connector specific directory. CONNDIR is defined
RED by caller
cd /d %CONNDIR%

REM set variables that need to pass to start_adapter.bat
REM set JVMArgs=
REM set JCLASSES=
REM set LibPath=
REM set ExtDirs=

REM A sample to start a C++ connector
REM call start_adapter.bat -nconnName -sServerName -dconnectorDLLfile -f...
REM -p... -c... ...

REM A sample to start a Java connector
call start_adapter.bat -nconnName -sserverName -lconnectorSpecificClasses
-f... -p... -c... ...

endlocal
```

Figure 71. Sample contents of the startup-script template for Windows

By convention, the start_*connName*.bat script has the standard syntax shown in Figure 72, with *connName* being the name of the connector, *ICSinstance* being the name of the InterChange Server instance, and *additionalOptions* specifies additional startup parameters to pass to the connector invocation. These options include -c, -f, -t, and -x. For more information, see Table 112 on page 219.

```
start_connName connName ICSinstance additionalOptions
```

Figure 72. Standard syntax for Windows connector startup script

As the connector developer, you control the content of start_*connName*.bat. Therefore, you can change the syntax of your connector startup script. However, if you change this standard syntax, make sure that all information that start_adapter.bat requires is available at the time of its invocation within start_*connName*.bat.

Note: In the start_*connName*.bat syntax in Figure 72, the *connName* and *ICSinstance* arguments are required. The *additionalOptions* argument is optional.

The startup script with the standard syntax makes the following assumptions about your connector's runtime files based on the connector name (*connName*):

- The connector name is the same as name of the connector directory under the connectors subdirectory of the product directory
- The connector name is the same as the Java connector's library file (its jar file, CW*connName*.jar), which resides in the connector directory

For example, for the MyJava connector to meet these assumptions, its runtime files must reside in the *ProductDir*\connectors\MyJava directory and its jar file must reside in that directory with the name BIA_MyJava.jar. If y our connector cannot meet these assumptions, you must customize its startup script to provide the appropriate information to the generic connector-invocation script, start_adapter.bat.

In this start_*connName*.bat file, take the following steps:

1. Call the `CWConnEnv.bat` environment file to initialize the startup environment.
2. Move into the connector directory.
3. Set the startup environment variables within the startup script with any connector-specific information and any connector-specific variables.
4. Call the `start_adapter.bat` script to invoke the connector.

The following sections describe each of these steps.

Calling the environment file: The `CWConnEnv.bat` file contains environment settings for the IBM Java Object Request Broker (ORB) and the IBM Java Runtime Environment (JRE). The following line invokes this environment file within the startup script:

```
call "%ADAPTER_RUNTIME%"\bin\CWConnEnv
```

Moving into the connector directory: The `start_`*`connName`*`.bat` script must change to the connector directory *before* it calls the `start_adapter.bat` script. The connector directory contains the connector-specific startup script as well as other files needed at connector startup. You can define the name of this connector directory any way you wish. However, as discussed in "Preparing the connector directory" on page 214, by convention the connector directory name matches the connector name.

If the `start_`*`connName`*`.bat` script uses the standard syntax (see Figure 72 on page 217), the connector name is passed in as the first argument (%1). In this case, the following lines move into the connector directory:

```
REM set the directory where the specific connector resides
set CONNDIR=%CROSSWORLDS%\connectors\%1

REM goto the connector specific drive & directory
cd /d %CONNDIR%
```

Alternatively, because the connector name is used in several components of the connector, you can define an environment variable to specify this connector name and then evaluate this environment variable for all subsequent uses of the connector name within the `start_`*`connName`*`.bat` script. The lines to set the environment variables for the connector name and connector directory could be as follows:

```
REM set the name of the connector
set CONNAME=%1

REM set the directory where the specific connector resides
set CONNDIR=%CROSSWORLDS%\connectors\%CONNAME%

REM goto the connector specific drive & directory
cd /d %CONNDIR%
```

Setting the environment variables: In the `start_`*`connName`*`.bat` script, you must provide any of the connector-specific information that the environment variables listed in Table 111 specify.

Table 111. Environment variables in the connector startup script

Variable name	Value
ExtDirs	Specify the location of any application-specific jar files.
JCLASSES	Specify any application-specific jar files. Jar files are separated with a semicolon (;).
JVMArgs	Add any arguments to be passed to the Java Virtual Machine (JVM).
LibPath	Specify any application-specific library paths.

The start_adapter.bat file uses the information in Table 111 as follows:

- It appends the JCLASSES and LibPath environment variables to the appropriate variables within the connector framework.
- It sets the external directories (java.ext.dirs) with the ExtDirs environment variable.
- It includes the JVMArgs environment variable in its list of arguments it passes to the JVM.

In addition to the environment variables in Table 111, you can also define your own connector-specific environment variables. Such variables are useful for information that can change from release to release. You can set the variable to a value appropriate for this release and then include the variable in the appropriate line of the startup script. If the information changes in the future, you only have to change the variable's value. You do not have to locate all lines that use this information.

Invoking the connector: To actually invoke the connector within the JVM, the start_*connName*.bat script must call the start_adapter.bat script. The start_adapter.bat script provides information to initialize the necessary environment for the connector runtime (which includes the connector framework) with its *startup parameters*. Therefore, you must provide the appropriate startup parameters to start_adapter.bat. Table 112 shows the startup parameters that the start_adapter.bat script recognizes.

Table 112. Startup parameters for start_adapter.bat script

Startup parameter	Description	Required?	Valid as additional command-line option to start_connName.bat?
-c*configFile*	The full path name of the connector's configuration file	Required if integration broker is other than ICS	Yes
-d*dllName*	The name of the C++ connector's library file (*dllName*), which is a dynamic link library (DLL). This DLL name should *not* include the .dll file extension.	Yes, for all C++ connectors	No
-f*pollFrequency*	The amount of time between polling actions. Possible *pollFrequency* values are: • The number of milliseconds between polling actions • key: causes the connector to poll only when you type the letter p in the connector's startup window. The key option must be specified in lowercase. • no: causes the connector not to poll. The no option must be specified in lowercase.	No Default is 1000 milliseconds	Yes
-j	Indicates that the connector is written in Java	No, as long as you specify the -l option for Java connectors	No
-l*classname*	The name of the Java connector's connector class (*className*)	Yes, for all Java connectors	No

Table 112. Startup parameters for start_adapter.bat script (continued)

Startup parameter	Description	Required?	Valid as additional command-line option to start_connName.bat?
-n*connectorName*	The name of the connector (*connectorName*) to start	Yes	No
-s*brokerName*	The name of the integration broker (*brokerName*) to which the connector connects	Yes	No
-t	Boolean value to turn off or on the connector property `SingleThreadAppCalls`, which guarantees that all calls the connector framework makes to the application-specific component are with one call-triggered flow. The default value is `false`.	No	Yes
-x*connectorProps*	Initializes the value of an application-specific connector property. Use the following format for each property you specify: *propName=value*	No	Yes

Make sure that the call to `start_adapter.bat` includes the following startup parameters:

- All required startup parameters:
  - To specify the name of the connector definition: `-n`

    If the name of the connector is passed in as the first argument (%1) to the `start_`*`connName`*`.bat` script (see Figure 72 on page 217), the `-n` startup parameter can be specified as follows:

    ```
    -n%1Connector
    ```

    If you define an environment variable for the connector name (such as `CONNAME`), this `-n` parameter could appear as follows:

    ```
    -n%CONNAME%Connector
    ```

  - To specify the name of the InterChange Server instance: `-s`

    If the name of the ICS instance is passed in as the second argument (%2) to the `start_`*`connName`*`.bat` script (see Figure 72 on page 217), the `-s` startup parameter can be specified as follows:

    ```
    -s%2
    ```

Other integration brokers

When your integration broker is a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Integration Message Broker), or WebSphere Application Server, the -c option is also a required startup parameter.

- Language-specific startup parameters required for a Java connector:

  To specify connector-specific classes (or package): `-l`

  For example, if you follow the recommended naming conventions, the language-specific parameter for the Java connector name is `MyJava` would be:

  ```
  -lcom.crossworlds.connectors.MyJava.MyJavaAgent
  ```

  If you define an environment variable for the connector name (such as `CONNAME`), this `-l` parameter could appear as follows:

  ```
  -lcom.crossworlds.connectors.%CONNAME%.%CONNAME%Agent
  ```

- Any optional startup parameters that apply to all invocations of your connector. Consult Table 112 on page 219. for a list of optional startup parameters.

The syntax for the call to start_adapter.bat should have the following format:

```
call start_adapter.bat -nconnName -sICSinstance languageSpecificParams
 -cCN_connNameConnector.cfg
 -...
```

For example, the following line invokes the MyJava connector:

```
call start_adapter.bat -lcom.crossworlds.connectors.MyJava.MyJavaAgent
    -nMyJava -sICSserver -cMyJavaConnector.cfg  -...
```

Note: The preceding command line assumes that the connector is running against an InterChange Server instance whose name is ICSserver. If the connector runs against an instance of WebSphere MQ Integrator Broker or WebSphere Message Broker, that instance name would need to appear in the command line.

With the use of the CONNAME environment variable to hold the connector name, this call can be generalized to the following:

```
call start_adapter.bat -n%CONNAME% -s%2 languageSpecificParams
 -cCN_%CONNAME%Connector.cfg
 -...
```

For the call to start_adapter.bat, keep the following points in mind:

- Make sure that the line to invoke the connector runtime is all *on one line* in your startup script; that is, no carriage returns should exist at the line breaks shown in the sample startup line.
- The order of the parameters listed in the call to start_adapter.bat is *not* important.
- You might also want to have your call to start_adapter.bat handle any additional options that the user might pass into the call to start_*connName*.bat. In this case, you should provide "extra" arguments to pass to start_adapter.bat so that additional options are passed down to the actual connector invocation. For example, the following call to start_adapter.bat handles up to three additional command-line options:

  ```
  call start_adapter.bat -n%CONNAME% -s%2 languageSpecificParams
   -cCN_%CONNAME%Connector.cfg %3 %4 %5
  ```

Creating the shortcut: A shortcut enables a connector to be started from a menu option within Programs > IBM WebSphere Business Integration Adapters > Adapters > Connectors. The shortcut should list the call to the start_*connName*.bat script. If this script uses the standard syntax (see Figure 72 on page 217), the shortcut would have the following form:

```
ProductDir\connectors\start_connName connName ICSinstance
```

If you define your own syntax for your start_*connName*.bat script, you must ensure that the shortcut uses this custom syntax.

If your menu already contains a shortcut for a Java connector that uses the start_*connName*.bat startup script, an easy way to create a shortcut is to copy this existing connector's shortcut and edit the shortcut properties to change the connector name or add any other startup parameters.

For example, for the MyJavaconnector that uses the standard syntax for its startup script, you could create the following shortcut:

```
ProductDir\bin\start_MyJava.bat MyJava ICSinstance
```

Note: The preceding command line assumes that the connector is running against an InterChange Server instance whose name is ICSinstance. If the connector runs against a WebSphere MQ Integrator Broker instance, that instance name would appear in the shortcut command line.

Startup script on UNIX systems

Starting a connector on a UNIX-based system involves the following steps:

1. Call the connector's startup script, connector_manager_*connName* with its -start option.

   The connector_manager_*connName* script (where *connName* is the name of your connector) is a connector-specific startup script. It identifies the name of the connector and provides the action to take on this connector with one of its options, which include -start and -stop. This script is generated with the Connector Script Generator tool. Once generated, the script resides in the bin subdirectory of the product directory. It is this connector_manager_*connName*.bat script that the user invokes to start the connector on a UNIX-based system.

2. Call the generic connector manager script, connector_manager.

   The connector_manager file is generic to all connectors. It generates the call to the connector-specific invocation script, start_*connName*.sh. the actual invocation of the connector within the JVM. It resides in the bin subdirectory of the product directory. The connector_manager_*connName* script calls the connector_manager script.

3. Call the connector-specific invocation script, start_*connName*.sh

   The start_*connName*.bat script provides connector-specific information (such as application-specific libraries and their locations). By convention, this script resides in the connector directory:

   ProductDir/connectors/*connName*

   The connector_manager script calls the start_*connName*.sh script to actually prepare the connector-specific information for connector invocation.

4. Call the generic connector-invocation script, start_adapter.sh

   The start_adapter.sh file is generic to all connectors. It performs the actual invocation of the connector within the JVM. It resides in the bin subdirectory of the product directory. The start_*connName*.sh script must call the start_adapter.sh script to actually invoke the connector.

Figure 73 shows the steps to start a connector on a UNIX-based system.

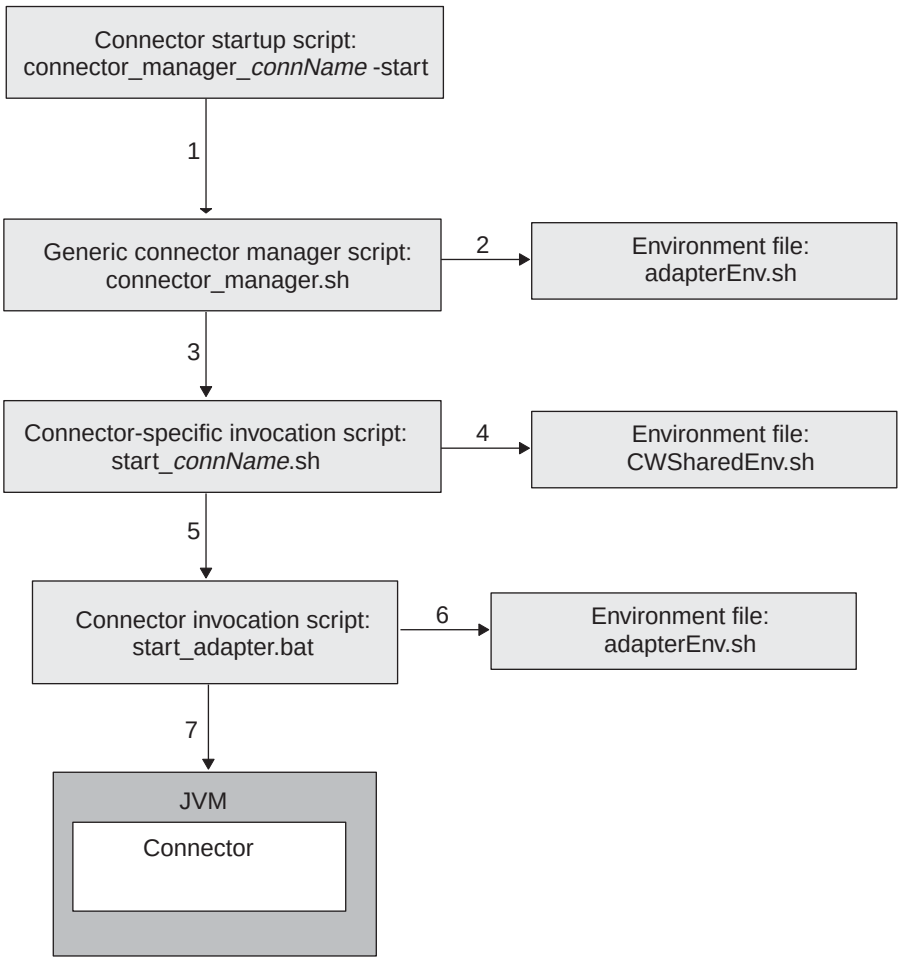

Figure 73. Starting a connector on a UNIX-based system

When a WebSphere Business Integration Adapters Installer installs connectors on a UNIX-based system, it takes the following steps:

- Install the generic `connector_manager` script and the generic `start_adapter.sh` connector invocation script in the `bin` subdirectory of the product directory.
- Install the `start_connName.sh` script in the `connectors/connName` subdirectory of the product directory.
- Generate the `connector_manager_connName` startup script, which is a wrapper for the generic `connector_manager` script. This generic script calls the appropriate `start_connName.sh` script, which begins the actual connector invocation.
- Install the new `connector_manager_connName` script in the `bin` product subdirectory.

  The `connector_manager_connName` script calls the `connector_manager` script, providing the appropriate command-line arguments, such as a local configuration file or a threading type.

In this sequence of steps, there are two scripts that are *not* generic; that is, no single script exists that can work with any connector:

- The `connector_manager_connName.sh` startup script is unique to each connector. However, it is generated by the installation process. Therefore, you do *not* need to create one for your custom connector.

- The custom invocation script, start_*connName*.sh, is also unique to each connector. Therefore, you must create a custom invocation script for your connector and put it in the connector*connName* subdirectory of the product directory.

Connector-specific connector-manager startup script: To start a connector, the connector_manager_*connName*.sh script has the syntax shown in Figure 74, with *connName* being the name of the connector and *additionalOptions* is an optional argument that specifies additional startup parameters to pass to the connector invocation. These options include -f and -x. For more information, see Table 113 on page 226.

```
connector_manager_connName -start additionalOptions
```

Figure 74. Syntax for starting a UNIX connector

To create a connector-specific connector-manager startup script, connector_manager_*connName*, you can use the Connector Script Generator tool (ConnConfig.sh in the product bin directory). Once you specify the connector name (*connName*), this tool generates the connector_manager_*connName* startup script and puts it in the bin subdirectory of the product directory. For information on this tool, see Appendix C, "Connector Script Generator," on page 511.

Connector-specific invocation script: To create a connector-specific invocation script, you create a new connector-specific script called start_*connName*.sh (where *connName* is your Java connector name). For example, if your Java connector has a connector name of MyJava, its startup script name is start_MyJava.sh. As a starting point, you can copy the startup-script template, which is located in the following file:

ProductDir/templates/start_connName.sh

Figure 75. shows a sample of the contents of the invocation-script template for UNIX. Please consult the version of this file released with your product for the most current contents.

```
#!/bin/sh

# set environment
\
#.${ADAPTER_RUNTIME}/bin/wbia_connEnv.sh

# If required, go to directory where connector class files reside
cd /
cd "${CONNDIR}"

# Please define the following variables that need to pass to callee
export JCLASSES=
export LibPath=
export ExtDirs=
export JVMArgs=

# Call base script start_adapter.sh to start a C++ connector
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -nconnName -sserverName
-dconnSpecificDLLfile -f... -p... -c... ...

# Call base script start_adapter.sh to start a Java connector
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -nconnName -sserverName
-lconnSpecificClasses -f... -p... -c... ...
```

Figure 75. Sample contents of the startup-script template for UNIX-based systems

On UNIX-based systems, the start_*connName*.sh script has the syntax shown in Figure 72 on page 217. However, unlike the start_*connName* script on Windows systems, this syntax for start_*connName* on UNIX-based systems *must* follow that shown in Figure 72 on page 217. The connector_manager script calls start_*connName* with this syntax. As the connector developer, you control the content of start_*connName*.sh but you should not change the syntax of this script.

If you follow the suggested naming conventions (see Table 109 on page 209), your connector-specific invocation script can make the following assumptions about your connector's runtime files based on the connector name (*connName*):

- The connector name is the same as name of the connector directory under the connectors subdirectory of the product directory
- The connector name is the same as the Java connector's library file (its jar file, CW*connName*.jar), which resides in the connector directory

For example, for the MyJava connector to meet these assumptions, its runtime files must reside in the *ProductDir*/connectors/MyJava directory and its jar file must reside in that directory with the name BIA_MyJava.jar. If y our connector cannot meet these assumptions, you must customize its startup script to provide the appropriate information to the generic connector-invocation script, start_adapter.sh.

In this start_*connName*.sh file, take the following steps:

1. Call the CWConnEnv.sh environment file to initialize the startup environment.
2. Move into the connector directory.
3. Set the startup environment variables within the startup script with any connector-specific information and any connector-specific variables.
4. Call the start_adapter.sh script to invoke the connector.

The following sections describe each of these steps.

Calling the environment file: The CWConnEnv.sh file contains environment settings for the IBM Java Object Request Broker (ORB) and the IBM Java Runtime Environment (JRE). The following line invokes this environment file within the startup script:

```
. ${ADAPTER_RUNTIME}/bin/CWConnEnv.sh
```

Moving into the connector directory: The start_*connName*.sh script must change to the connector directory *before* it calls the start_adapter.sh script. The connector directory contains the connector-specific startup script as well as other files needed at connector startup. You can define the name of this connector directory any way you wish. However, as discussed in "Preparing the connector directory" on page 214, by convention the connector directory name matches the connector name.

The start_*connName*.sh script expects the connector name to be passed in as the first argument ($1). Therefore, the following lines move into the connector directory:

```
# set the directory where the specific connector resides
CONNDIR=${CROSSWORLDS}/connectors/$1
export CONNDIR

# If required, go to directory where connector class files reside
cd /
cd "${CONNDIR}"
```

Alternatively, because the connector name is used in several components of the connector, you can define an environment variable to specify this connector name and then evaluate this environment variable for all subsequent uses of the connector name within the `start_connName.sh` script. The lines to set the environment variables for the connector name and connector directory could be as follows:

```
# set the name of the connector
CONNAME=$1
export CONNNAME

REM set the directory where the specific connector resides
CONNDIR=${CROSSWORLDS}/connectors/${CONNAME}
export CONNDIR

# If required, go to directory where connector class files reside
cd /
cd "${CONNDIR}"
```

Setting the environment variables: In the `start_connName.sh` script, you must specify any of the connector-specific information that the environment variables listed in Table 111 on page 218. The `start_adapter.sh` script uses these environment variables in the same way as the `start_adapter.bat` script does on Windows systems. You can also define your own connector-specific environment variables for information that can change from release to release. For more information, see "Setting the environment variables" on page 218.

Invoking the connector: To actually invoke the connector within the JVM, the `start_connName.sh` script must call the `start_adapter.sh` script. The `start_adapter.sh` script provides information to initialize the necessary environment for the connector runtime (which includes the connector framework) with its *startup parameters*. Therefore, you must provide the appropriate startup parameters to `start_adapter.sh`. Table 113. Table 113 shows the startup parameters that the `start_adapter.sh` script recognizes.

Table 113. Startup parameters for start_adapter.sh script

Startup parameter	Description	Required?	Valid as additonal command-line option for connector_manager_connName?
-b	Runs the connector as a background thread; that is, the connector does *not* receive any input from standard input (STDIN). The generic `connector_manager` script (called by each `connector_manager_connName` script) automatically specifies this option when it invokes the `start_connName.sh` script. Therefore, to prevent a connector from being run in the background, you can remove the `-b` parameter from the `start_connName.sh` invocation.	See the description	No
-c*configFile*	The full path name of the connector's configuration file	Required if integration broker is other than ICS	Yes

Table 113. Startup parameters for start_adapter.sh script (continued)

Startup parameter	Description	Required?	Valid as additonal command-line option for connector_manager_connName?
-f*pollFrequency*	The amount of time between polling actions. Possible *pollFrequency* values are: • The number of milliseconds between polling actions • `key`: causes the connector to poll only when you type the letter p in the connector's startup window. The `key` option must be specified in lowercase. • `no`: causes the connector not to poll. The `no` option must be specified in lowercase. The value that the `-f` parameter specifies overrides the polling frequency in the connector's configuration file.	No Default is 1000 milliseconds	Yes
-l*classname*	The name of the Java connector's connector class (*className*) **Note:** The `-b` parameter is *not* a valid command-line option for the `connector_manager_`*`connName`* script.	Yes	No
-n*connectorName*	The name of the connector (*connectorName*) to start	Yes	No
-s*brokerName*	The name of the integration broker (*brokerName*) to which the connector connects	Yes	No
-t*threadingType*	Specifies the threading model to use for the connector. Possible values for *threadingType* are: • SINGLE_THREADED: only a single thread accesses the application. • MAIN_SINGLE_THREADED: only the main thread accesses the application. • MULTI_THREADED: multiple threads can access the application	No	No
-x*connectorProps*	Initializes the value of an application-specific connector property. Use the following format for each property you specify: *`propName=value`*	No	Yes

Make sure that the call to `start_adapter.sh` includes the following startup parameters:

- All required startup parameters:
  - To specify the name of the connector definition: `-n`

    Because the name of the connector is passed in as the first argument ($1) to the `start_`*`connName`*`.sh` script (see Figure 72 on page 217), the `-n` startup parameter can be specified as follows:

    ```
    -n${1}Connector
    ```

    If you define an environment variable for the connector name (such as `CONNAME`), this `-n` parameter could appear as follows:

```
-n${CONNAME}Connector
```

– To specify the name of the InterChange Server instance: `-s`

  If the name of the ICS instance is passed in as the second argument ($2) to the `start_connName.sh` script (see Figure 72 on page 217), the `-s` startup parameter can be specified as follows:

```
-s${2}
```

Note: All UNIX connectors usually include the `-b` startup parameter so that the connector process runs in the background. Therefore, the `connector_manager` generic startup script automatically specifies this startup parameter for *all* connectors. You do *not* need to specify it in the `start_adapter.sh` call.

Other integration brokers

When your integration broker is WebSphere MQ Integrator Broker, WebSphere Integration Message Broker, or WebSphere Application Server, the -c option is also a required startup parameter.

- Language-specific startup parameters required for a Java connector:

  To specify connector-specific classes (or package): `-l`

  For example, if you follow the recommended naming conventions, the language-specific parameter for the Java connector name is `MyJava` would be:

```
-lcom.crossworlds.connectors.MyJava.MyJavaAgent
```

  If you define an environment variable for the connector name (such as `CONNAME`), this `-l` parameter could appear as follows:

```
-lcom.crossworlds.connectors.${CONNAME}.${CONNAME}Agent
```

- Any optional startup parameters that apply to all invocations of your connector. Consult Table 113 on page 226. for a list of optional startup parameters.

For more information about the startup parameters, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set or your implementation guide in the WebSphere Business Integration Adapters documentation set.

The syntax for the call to `start_adapter.sh` should have the following format:

```
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -nconnDefName -sICSinstance
 -lclassName -cCN_connNameConnector.cfg
 -...
```

For example, the following line invokes the `MyJava` connector:

```
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -nMyJavaConnector -sICSserver
 -lcom.crossworlds.connectors.MyJava.MyJavaAgent
 -cMyJavaConnector.cfg  -...
```

Note: The preceding command line assumes that the connector is running against an InterChange Server instance whose name is `ICSserver`. If the connector runs against a WebSphere MQ Integrator Broker instance, that instance name would need to appear in the command line.

With the use of the `CONNAME` environment variable to hold the connector name, this call can be generalized to the following:

```
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -n${CONNAME}Connector -s${2}
-lclassName -cCN_${CONNAME}Connector.cfg  -...
```

For the call to `start_adapter.sh`, keep the following points in mind:

- Make sure that the line to invoke the connector runtime is all *on one line* in your startup script; that is, no carriage returns should exist at the line breaks shown in the sample startup line.
- The order of the parameters listed in the call to `start_adapter.sh` is *not* important.
- You might also want to have your call to `start_adapter.sh` handle any additional options that the user might pass into the call to `connector_manager_`*`connName`*`.sh` (see Figure 74 on page 224.).. In this case, you should provide "extra" arguments to pass to `start_adapter.sh` so that additional options are passed down to the actual connector invocation. For example, the following call to `start_adapter.sh` handles three additional command-line options:

```
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -n${CONNAME}Connector -s${2}
-l className -cCN_${CONNAME}Connector.cfg ${3} ${4} ${5}
```

Starting a connector as a Windows service

You can set up a connector to run as a Windows service that can be started and stopped by a remote administrator. For more information, see the *System Installation Guide for Windows* in the IBM WebSphere InterChange Server documentation set or your implementation guide in the IBM WebSphere Business Integration Adapter documentation set.

Note: If you are using InterChange Server as your integration broker and you want to use the automatic-and-remote restart feature with the connector, do *not* start connector as a Windows service. Instead, start the MQ Trigger Monitor as a service. For more information, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Part 3. Java connector library API reference

Chapter 9. Overview of the Java connector library

The Java connector library include class libraries that you need to use when developing a connector. This connector class library contains predefined classes for connectors in Java. You use these class libraries to derive connector classes and methods. The class libraries also provide utilities, such as methods to implement tracing and logging services.

IBM provides a Java jar file (Java archive file), `WBIA.jar`, that contains the predefined classes and interfaces of the Java connector library. The current version of the `WBIA.jar` file resides in the `lib` subdirectory of the product directory. Older versions of the `WBIA.jar` file reside in the following product subdirectory:

`lib\WBIA\`*`version`*

where *`version`* is the version of the Java connector library. The current version of `WBIA.jar` is compatible with older versions of this library.

Note: For instructions on building a Java connector to run on Windows NT or Windows 2000, see "Compiling the connector" on page 210.

Classes and interfaces

Table 114 lists the classes and interfaces in the Java connector library.

Table 114. Classes and interfaces in the Java connector library

Class or interface	Description	Page
`CWConnectorAgent`	Represents the base class for a connector. You extend this class to define your connector class and implement the required virtual methods	449
`CWConnectorAttrType`	Defines the attribute-type constants	249
`CWConnectorBOHandler`	Represents the base class for a business object handler. You extend this class to define one or more business object handler for your connector.	251
`CWConnectorBusObj`	Represents a business object instance. It provides access to the business object, business object definition, and the attributes	257
`CWConnectorConstant`	Defines constants for use with the Java connector library: • outcome-status constants • verb constants	305
`CWConnectorEvent`	Represents an event object, which holds information from an event record that has been retrieved from an event store	307
`CWConnectorEventStatusConstants`	Defines event-status constants, which represent the status values that an event record can have	315
`CWConnectorEventStore`	Represents an event store, which holds event records for access by the connector's event detection mechanism (usually polling)	319
`CWConnectorEventStoreFactory`	Represents the event-store factory, which instantiates a `CWConnectorEventStore` object	333
`CWConnectorExceptionObject`	Represents an exception-detail object, which contains additional status information that is included in an exception object	335

Table 114. Classes and interfaces in the Java connector library (continued)

Class or interface	Description	Page
`CWConnectorLogAndTrace`	Defines constants for use with logging and tracing services: • message-file constants • message-type constants • trace-level constants	341
`CWConnectorReturnStatusDescriptor`	Represents a return-status descriptor, which contains error and informational messages	343
`CWConnectorUtil`	Provides miscellaneous utility methods for use in a Java connector; These utility methods fall into the following general categories: • Static methods for generating and logging messages • Static methods for creating business objects • Static methods for obtaining connector configuration properties • Methods for obtaining locale information	347
`CWException`	Represents an exception object for the Java connector library	383
`CWProperty`	Represents a connector-property object, which contains a hierarchical connector configuration property	449

Chapter 10. CWConnectorAgent class

The CWConnectorAgent class is the base class for a Java connector. From this class, a connector developer must derive a *connector class* and implement the user-defined methods for the connector. This derived connector class contains the code for the application-specific component of the connector.

Note: The CWConnectorAgent class extends the ConnectorBase class of the low-level Java connector library. For more information on the classes of the low-level Java connector library, see Chapter 26, "Overview of the low-level Java connector library," on page 405.

Important: All Java connectors *must* extend this connector base class and provide implementations for the following methods: agentInit(), getVersion(), getConnectorBOHandlerForBO(), pollForEvents(), and terminate(). However, CWConnectorAgent provides default implementations for the getVersion(), getConnectorBOHandlerForBO(), and pollForEvents() methods. In their derived connector base class, developers can either use these default implementations or override them to implement their own versions. Developers *must* provide implementations for the agentInit() and terminate() methods.

Table 115 summarizes the methods in the CWConnectorAgent class.

Table 115. Member methods of the CWConnectorAgent class

Member method	Description	Page
CWConnectorAgent()	Creates a connector object.	235
agentInit()	Initializes the connector	236
executeCollaboration()	Sends business object requests to collaborations as a synchronous request.	238
getCollabNames()	Retrieves the list of collaborations that are available to process business object requests.	239
getConnectorBOHandlerForBO()	Retrieves the business object handler for a specified business object definition.	239
getEventStore()	Retrieves a reference to the connector's event store.	240
getVersion()	Retrieves the version of the connector.	241
gotApplEvent()	Sends a business object to InterChange Server.	242
isAgentCapableOfPolling()	Determines whether this connector process is capable of polling.	243
isSubscribed()	Determines whether the integration broker has subscribed to a particular business object with a particular verb.	245
pollForEvents()	Polls an application's event store for events that cause changes to business objects.	246
terminate()	Closes the connection with the application and frees allocated resources.	247

CWConnectorAgent()

Creates a connector object.

Syntax

```
public CWConnectorAgent();
```

Parameters

None.

Return values

A `CWConnectorAgent` object containing the newly created connector.

agentInit()

Initializes the connector.

Syntax

```
public void agentInit();
```

Parameters

None.

Return values

None.

Exceptions

`ConnectionFailureException`
: Thrown if the connector fails to obtain a connection with the application.

`InProgressEventRecoveryFailedException`
: Thrown if the connector is unable to perform in-progress event recovery.

`LogonFailedException`
: Thrown if the connector is unable to log into the application.

`PropertyNotSetException`
: Thrown if the connector retrieves any required connector configuration property that does not have a value set for it.

Notes

The `agentInit()` method performs all initialization functionality for the connector, including any of the following tasks required for the connector's application-specific component:

- Establishing a connection
- Retrieving connector properties
- Recovering In-Progress events

Important: The `CWConnectorAgent` class does not provide a default implementation for the `agentInit()` class. Therefore, the connector class *must* implement this method.

The connector framework calls the `agentInit()` method to initialize the connector when it comes up. If `agentInit()` performs any of the conditions listed in

Table 116,, it *must* check for the following conditions and throw the appropriate exception.

Table 116. Exceptions to throw from the agentInit() method

Condition	Exception to throw
If the connector retrieves any required connector configuration property that is not set	`PropertyNotSetException`
If the connector fails to obtain a connection with the application	`ConnectionFailureException`
If the connector fails to log onto the application	`LogonFailedException`
If the `recoverInProgressEvents()` method finds In-Progress events in the event store and some failure occurs during the recovery process	`InProgressEventRecoveryFailedException`

To throw one of the exceptions in Table 116, take the steps outlined in Table 117:

Table 117. Handling an initialization error

Error-handling step	Method or code to use
1. If an error has occurred, log an error message to the log destination to indicate the cause of the initialization error.	`CWConnectorUtil.generateAndLogMsg()`
2. Instantiate an exception-detail object to hold the exception information.	`CWConnectorExceptionObject` *`excptnDtailObj`* `= new CWConnectorExceptionObject();`
3. Set the status information within an exception-detail object:	
• set a message to indicate the cause of the initialization failure	*`excptnDtailObj`*`.setMsg()`
• set the status to an outcome status that tells the connector framework the success of the initialization. If you want the initialization process (and the connector) to terminate, set the outcome status to `CxConnectorConstant.FAIL`.	*`excptnDtailObj`*`.setStatus()`
4. Throw the `agentInit()` exception from Table 116 that indicates the initialization failure. This exception is how the `agentInit()` method tells the connector framework that a initialization error has occurred. This exception object contains the exception-detail object you initialized in Step 3. When the low-level `init()` method (which calls `agentInit()`) catches this exception object, it copies the status from the exception-detail object into its own return status, which it returns to the connector framework. **Note:** If you do not set the exception status within the exception-detail object, the `init()` method returns an outcome status of `FAIL` and the connector framework terminates the connector.	`throw new` *`agentInit`*`Exception(` *`excptnDtailObj`*`);`

See also

generateAndLogMsg(), recoverInProgressEvents()

executeCollaboration()

Sends a business object request to the connector framework, which sends it to a business process within the integration broker. This is a synchronous request.

Syntax

```
public void executeCollaboration(String busProcName,
   CWConnectorBusObj theBusObj,
   CWConnectorReturnStatusDescriptor rtnStatusDesc);
```

Parameters

busProcName	Specifies the name of the business process to execute the business object request. If InterChange Server is your integration broker, the business-process name is the name of a collaboration.
theBusObj	Is the triggering event and the business object returned from the business process.
rtnStatusDesc	Is the return-status descriptor containing a message and the execution or return status from the business process.

Return values

None.

Exceptions

None.

Notes

The `executeCollaboration()` method sends the *theBusObj* business object to the connector framework. The connector framework does some processing on the event object to serialize the data and ensure that it is persisted properly. It then sends the event to the *busProcName* business process in the integration broker. This method initiates a synchronous execution of an event, which means that the method waits for a response from the integration broker's business process.

WebSphere InterChange Server

If your integration broker is IBM WebSphere InterChange Server, the business process that `executeCollaboration()` invokes is a collaboration.

To receive status information about the business-process execution, pass in an instantiated return-status descriptor, *rtnStatusDesc*, as the last argument to the method. The integration broker can return status information from its business process and send it to the connector framework, which populates this return-status descriptor with it. You can use the methods of the `CWConnectorReturnStatusDescriptor` class to access this status information.

Note: To initiate an asynchronous execution of an event, use the `gotApplEvent()` method. Asynchronous execution means that the calling code does *not* wait for the receipt of the event, nor does it wait for a response.

See also

gotApplEvent(),, methods of the `CWConnectorReturnStatusDescriptor` class

getCollabNames()

Retrieves the list of collaborations that are available to process business object requests.

WebSphere InterChange Server
This method is only valid when the integration broker is InterChange Server.

Syntax

```
public String[] getCollabNames();
```

Parameters

None.

Return values

An array of `String` objects containing a list of collaboration names.

Exceptions

None.

getConnectorBOHandlerForBO()

Retrieves the business object handler for a specified business object definition.

Syntax

```
public CWConnectorBOHandler getConnectorBOHandlerForBO(
   String busObjName);
```

Parameters

busObjName Is the name of a business object.

Return values

A reference to a `CWConnectorBOHandler` object, which represents the business object handler for the *busObjName* business object.

Exceptions

None.

Notes

The connector framework calls the `getConnectorBOHandlerForBO()` method to retrieve the business object handler for a business object definition. You can use one business object handler for multiple business object definitions or a business object handler for each business object definition.

Important: The CWConnectorAgent class provides a default implementation for the getConnectorBOHandlerForBO() method. Therefore, you can either use this default implementation or override the method to return your own business-object-handler class.

The CWConnectorAgent class provides a default implementation for the getConnectorBOHandlerForBO() method, which returns a reference to a business object handler of the ConnectorBOHandler class. To use this default implementation, you would extend the CWConnectorBOHandler class, naming this extended class ConnectorBOHandler. If you name your business-object-handler base class something other than ConnectorBOHandler, you must override getConnectorBOHandlerForBO() to return a reference to your extended business-object-handler base class.

---

getEventStore()

Creates a reference to the connector's event store.

Syntax

```
public CWConnectorEventStore getEventStore();
```

Parameters

None.

Return values

A CWConnectorEventStore object that provides access to the connector's event store. If the event-store-factory class cannot be located, the method returns null.

Exceptions

None.

Notes

The getEventStore() method is the event-store factory, whose task is to instantiate an event-store object for the connector. Through this event-store object, the connector can access its event store. The getEventStore() method calls the getEventStore() method of your event-store-factory class, which implements the CWConnectorEventStoreFactory interface.

Important: The CWConnectorAgent class provides a default implementation for the getEventStore() method. Therefore, you can either use this default implementation or override the method to implement your own mechanism to instantiate an event-store object.

The default implementation of the getEventStore() method that the CWConnectorAgent class provides checks the EventStoreFactory connector configuration property for the name of the event-store-factory class (which implements the CWConnectorEventStoreFactory interface), as follows:

- If the EventStoreFactory property is set, getEventStore() instantiates the specified event-store-factory class and calls its getEventStore() method to return an event-store object.
- If the EventStoreFactory property is *not* set, getEventStore() tries to build the name of the event-store-factory class.

From the name of the connector package, the getEventStore() method extracts the connector name. It assumes that the event store is named as follows:

*connectorName*EventStore

For example, for the WebSphere Business Integration Adapter for JDBC, the connector name is JDBC. Therefore, the getEventStore() would generate JDBCEventStore as the name of the connector's event store and try to instantiate an event-store-factory class of this name.

The EventStoreFactory property must specify the entire class name for the event-store factory instance. For information on the format of this property, see "CWConnectorEventStoreFactory interface" on page 178. For example, the WebSphere Business Integration Adapter for JDBC contains an event-store factory that provides access to a JDBC event store. Therefore, the EventStoreFactory property might be set as follows:

```
com.crossworlds.connectors.JDBC.JDBCEventStoreFactoryInstance
```

The default implementation of the pollForEvents() method calls this getEventStore() method to retrieve a reference to the event store. For more information, see "Retrieving event records" on page 182.

See also

getEventStore(),, pollForEvents()

---

getVersion()

Retrieves the version of the connector.

Syntax

```
public String getVersion();
```

Parameters

None.

Return values

A String indicating the version of the connector's application-specific component.

Exceptions

None.

Notes

The connector framework calls the getVersion() method to retrieve the version of the connector. The getVersion() methods is usually called as part of the connector initialization process, from within the agentInit() method. The connector framework also calls the getVersion() method to get a version for the connector.

Important: The CWConnectorAgent class provides a default implementation for the getVersion() method. Therefore, you can either use this default implementation or override the method to implement your own versioning mechanism.

The CWConnectorAgent class provides a default implementation for the getVersion() method, which retrieves the package name from standard class information. It then gets the version from the manifest file present in the package.

gotApplEvent()

Sends a business object request to the connector framework. This is an asynchronous request.

Syntax

```
public int gotApplEvent(CWConnectorBusObject theBusObj);
```

Parameters

theBusObj Is the business object instance being sent to the connector framework.

Return values

An integer that indicates the outcome status of the event delivery. Compare this integer value with the following outcome-status constants to determine the status:

CWConnectorConstant.SUCCEED
: The connector framework successfully delivered the business object to the connector framework.

CWConnectorConstant.FAIL
: The event delivery failed.

CWConnectorConstant.CONNECTOR_NOT_ACTIVE
: The connector is paused and therefore unable to receive events.

CWConnectorConstant.NO_SUBSCRIPTION_FOUND
: No subscriptions for the event that the business object represents.

Exceptions

None.

Notes

The gotApplEvent() method sends the *theBusObj* business object to the connector framework. The connector framework does some processing on the event object to serialize the data and ensure that it is persisted properly. It then makes sure the event is sent to the integration broker.

WebSphere InterChange Server

If the integration broker is InterChange Server, the connector framework sends the event (as a business object) to InterChange Server across its configured delivery transport mechanism (such as JMS or CORBA IIOP).

Other integration brokers

If the integration broker is a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework sends the event (as an XML message) to the integration broker across its configured delivery transport mechanism of a JMS queue.

Before sending the business object to the connector framework, `gotApplEvent()` checks for the following conditions and returns the associated outcome status if these conditions are *not* met:

Condition	Outcome status
Is the status of the connector active; that is, it is not in a "paused" state? When the connector's application-specific component is paused, it no longer polls the application.	`CONNECTOR_NOT_ACTIVE`
Is there a subscription for the event?	`NO_SUBSCRIPTION_FOUND`

Note: Because `gotApplEvent()` makes sure that the business object and verb to be sent have a valid subscription, you do *not* need to call `isSubscribed()` immediately before calling `gotApplEvent()`.

The connector uses the `pollForEvents()` method to poll the event store for subscribed events to send to the integration broker. Within `pollForEvents()`, the connector uses the `gotApplEvent()` method to send an event (represented as a business object) to the connector framework. The connector framework then routes this business object to the integration broker. Therefore, the poll method should check the return code from `gotApplEvent()` to ensure that any errors that are returned are handled appropriately. For example, until the event delivery is successful, the poll method should *not* remove the event from the event store. Instead, the poll method should update the event record's status to reflect the results of the event delivery based on the return code of `gotApplEvent()`. For more information, see "Sending the business object" on page 189.

The `gotApplEvent()` method initiates an asynchronous execution of an event. Asynchronous execution means that the method does *not* wait for receipt of the event, nor does it wait for a response.

Note: To initiate a synchronous execution of an event, use the `executeCollaboration()` method. Synchronous execution means that the calling code waits for the receipt of the event, and for a response.

See also

executeCollaboration(),, isSubscribed(),, pollForEvents()

isAgentCapableOfPolling()

Determines whether this connector process is capable of polling.

WebSphere InterChange Server
This method is only valid when the integration broker is InterChange Server.

Syntax

```
boolean isAgentCapableOfPolling();
```

Parameters

None.

Return values

A `boolean` value that indicates whether this connector is capable of polling. This return value depends on the type of connector:

Connector process type	Return value
Master (serial processing)	`true`
Master (parallel processing)	`false`
Slave (request)	`false`
Slave (polling)	`true`

Exceptions

None.

Notes

If a connector is configured to run in the single-process mode (with `ParallelProcessDegree` set to 1, which is the default), the `isAgentCapableOfPolling()` method always returns `true` because the same connector process performs both event polling and request processing.

If a connector is configured to run in parallel-process mode (with `ParallelProcessDegree` greater than 1), it consists of several processes, each with a particular purpose, as shown in Table 118.

Table 118. Purposes of processes of a parallel connector

Connector process	Purpose of connector process
Connector-agent master process	Receives the incoming event from ICS and determines to which of the connector's slave processes to route the event
Request-processing slave process	Handles requests for the connector
Polling slave process	Handles polling and event delivery for the connector

The return value of `isAgentCapableOfPolling()` depends on the purpose of the connector-agent process that makes the call to this method. For a parallel-process connector, this method returns `true` *only* when called from a connector whose purpose is to serve as a polling slave. For more information on parallel-process connectors, see the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set.

Note: Because the `isAgentCapableOfPolling()` method obtains information about the parallel-process mode of a connector, it must run against a version of

InterChange Server (ICS) that supports this feature. Therefore, to behave as documented here, `isAgentCapableOfPolling()` must run against a version 4.0 or later ICS. If run against an earlier version of ICS, `isAgentCapableOfPolling()` always returns `true`.

isSubscribed()

Determines whether the integration broker has subscribed to a particular business object with a particular verb.

Syntax

```
public boolean isSubscribed(String busObjName, String verb);
```

Parameters

busObjName Is the name of a business object for which subscriptions are to checked.

verb Is the active verb for the business object.

Return values

Returns `true` if the integration broker is interested in receiving the specified business object and verb; otherwise, returns `false`.

Exceptions

None.

Notes

The `isSubscribed()` method is part of the subscription manager, which tracks all subscribe and unsubscribe messages that arrive from the connector framework and maintains a list of active business object subscriptions. For a Java connector, this subscription manager is part of the connector base class.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the poll method can determine if any collaboration subscribes to the *busObjName* business object with the specified *verb*. At initialization, the connector framework requests its subscription list from the connector controller. At runtime, the poll method can use `isSubscribed()` to query the connector framework to verify that some collaboration subscribes to a particular business object. The poll method can send the event only if some collaboration is currently subscribed. For more information, see "Business object subscription and publishing" on page 13.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework assumes that the integration broker is interested in *all* the connector's supported business objects. If the application-specific component uses the `isSubscribed()` method to query the connector framework about subscriptions for a particular business object, the method returns `true` for *every* business object that the connector supports.

See also

gotApplEvent(),, pollForEvents()

---

pollForEvents()

Polls an application's event store for events that cause changes to business objects.

Syntax

```
public int pollForEvents();
```

Parameters

None.

Return values

An integer that indicates the outcome status of the polling operation.The `pollForEvents()` method typically uses the following return codes:

`CWConnectorConstant.SUCCEED`
: The polling action succeeded regardless of whether an event is retrieved.

`CWConnectorConstant.FAIL`
: The polling operation failed.

`CWConnectorConstant.APPRESPONSETIMEOUT`
: The application is not responding.

Exceptions

None.

Notes

The connector framework calls the `pollForEvents()` method, at a time interval that you can configure, so that the connector can detect any event in the application that is interesting to a subscriber. The frequency at which the class library calls this method depends on the poll frequency value that is configured by the `PollFrequency` connector configuration property.

Note: The `CWConnectorAgent` class provides a default implementation for the `pollForEvents()` method. Therefore, you can either use this default

implementation or override the method to implement your own polling mechanism. To provide a different polling behavior, you can implement your own version of `pollForEvents()`.

The `CWConnectorAgent` class provides a default implementation for the `pollForEvents()` method, which is based on the `CWConnectorEvent` event objects as a standard interface for event management. For information on the behavior of this default implementation, see "Implementing the pollForEvents() method" on page 180.. This default implementation provides the basic steps for polling an event store. If you override the default `pollForEvents()`, your implementation must perform similar steps.

Note: If your connector executes in a parallel-process mode, it uses a separate polling slave process to handle polling.

See also

gotApplEvent(),, isSubscribed()

---

terminate()

Terminates the connector, performing any required clean-up tasks.

Syntax

```
public int terminate();
```

Parameters

None.

Return values

An integer that indicates the status value of the `terminate()` operation.

`CWConnectorConstant.SUCCEED`
: The terminate operation succeeded.

`CWConnectorConstant.FAIL`
: The terminate operation failed.

Exceptions

None.

Notes

The connector infrastructure calls the `terminate()` method when the connector is shutting down. In your implementation of this method, it is good practice to free all the memory and log off from the application. You must implement this method for the connector.

Important: The `CWConnectorAgent` class does not provide a default implementation for the `terminate()` method. Therefore, the connector class *must* implement this method if resource clean-up is required.

Chapter 11. CWConnectorAttrType class

The CWConnectorAttrType class is the attribute-type class for Java connectors. It defines static constants for data types of attributes in a business object definition.

Attribute-type constants

The CWConnectorAttrType class defines numeric and string equivalents for attribute types. Table 119 summarizes the attribute-type constants in the CWConnectorAttrType class.

Table 119. Static constants of the CWConnectorAttrType class

Attribute data type	Numeric attribute-type constant	String attribute-type constant
Boolean	BOOLEAN	BOOLSTRING
Business object: multiple cardinality	None	MULTIPLECARDSTRING
Business object: single cardinality	None	SINGLECARDSTRING
Ciphertext	CIPHERTEXT	CIPHERTEXTSTRING
″Missing ID″	None	CXMISSINGID_STRING
Date	DATE	DATESTRING
Double	DOUBLE	DOUBSTRING
Float	FLOAT	FLTSTRING
Integer	INTEGER	INTSTRING
Invalid data type	INVALID_TYPE_NUM	INVALID_TYPE_STRING
LongText	LONGTEXT	LONGTEXTSTRING
Object	OBJECT	None
String	STRING	STRSTRING
Blank value	None	CxBlank
Ignore value	None	CxIgnore

Chapter 12. CWConnectorBOHandler class

The CWConnectorBOHandler class is the base class for the business object handlers of a Java connector. It provides the code to implement and access one business object handler. From this class, a connector developer must derive business-object-handler classes (as many as needed) and implement the doVerbFor() method for the business object handler.

Note: The CWConnectorBOHandler class extends the BOHandlerBase class of the low-level Java connector library. For more information on the classes of the low-level Java connector library, see Chapter 26, "Overview of the low-level Java connector library," on page 405.

Important: All Java connectors *must* extend this class. The name ConnectorBOHandler is the default name for the derived business-object-handler class. Developers can either use this default name or choose a different name for the derived business-object-handler class. Regardless of the name of the class, developers *must* implement the single method, doVerbFor(), in their derived business-object-handler class. If your connector handles request processing, your doVerbFor() method must provide verb processing for all supported verbs for the business object (or objects) it handles. If your connector does *not* provide request processing, it must still provide verb processing for the Retrieve verb.

An connector includes one or more business object handlers to perform tasks for the verbs in business objects. Depending on the active verb, a business object handler can insert business object data into an application, retrieve data, delete application data, or perform another task. For an introduction to request processing and business object handlers, see "Request processing" on page 24. For information on how to implement a business object handler, see Chapter 4, "Request processing," on page 79.

Table 120 summarizes the methods in the CWConnectorBOHandler class.

Table 120. Member methods of the CWConnectorBOHandler class

Member method	Description	Page
CWConnectorBOHandler()	Creates a business-object-handler object.	251
doVerbFor()	Performs the verb processing for the active verb of a business object.	252
getName()	Retrieves the name of the business-object-handler object.	254
setName()	Sets the name of the business-object-handler object.	254

CWConnectorBOHandler()

Creates a business-object-handler object.

Syntax

```
public CWConnectorBOHandler();
```

Parameters

None.

Return values

A `CWConnectorBOHandler` object containing the newly created business-object-handler object.

Notes

The `CWConnectorBOHandler()` constructor creates an instance of the `CWConnectorBOHandler` class, to which business object definitions can refer for performing the tasks of verbs in business objects. Typically, a connector developer derives a class from `CWConnectorBOHandler` and implements the `doVerbFor()` method for this derived class. The developer can call the constructor of this derived class in the `getConnectorBOHandlerForBO()` method of the `CWConnectorAgent` class to instantiate one or more business object handlers.

See also

getConnectorBOHandlerForBO()

---

doVerbFor()

Performs the verb processing for the active verb of a business object.

Syntax

```
public int doVerbFor(CWConnectorBusObj theBusObj);
```

Parameters

theBusObj Is the business object whose active verb is to be processed.

Return values

An integer that indicates the outcome status of the verb operation. Compare this integer value with the following outcome-status constants to determine the status:

`CWConnectorConstant.SUCCEED`
The verb operation succeeded.

`CWConnectorConstant.FAIL`
The verb operation failed.

`CWConnectorConstant.APPRESPONSETIMEOUT`
The application is not responding.

`CWConnectorConstant.VALCHANGE`
At least one value in the business object changed.

`CWConnectorConstant.VALDUPES`
The requested operation found multiple records in the application database for the same key value.

`CWConnectorConstant.MULTIPLE_HITS`
The connector finds multiple matching records when retrieving using non-key values. The connector returns a business object only for the first matching record.

`CWConnectorConstant.RETRIEVEBYCONTENT_FAILED`
: The connector was not able to find matches for Retrieve by non-key values.

`CWConnectorConstant.BO_DOES_NOT_EXIST`
: The connector performed a Retrieve operation, but the application database does not contain a matching entity for the requested business object.

Exceptions

`ConnectionFailureException`
: Thrown if the connector has lost the connection with the application.

`VerbProcessingFailedException`
: Thrown if the verb processing fails.

Notes

The `doVerbFor()` method performs the action of the active verb in the *theBusObj* business object. This method is the primary public interface for the business object handler. However, when the connector framework invokes a business object handler, it actually executes the low-level `doVerbFor()` method, inherited from the `BOHandlerBase` class. The low-level `doVerbFor()` method calls this `doVerbFor()` (in the business-object-handler class), which the connector developer must implement. For more information, see "Populating the return-status descriptor" on page 170..

Important: The `CWConnectorBOHandler` class does not provide a default implementation of the `doVerbFor()` method. Therefore, the business-object-handler class must implement this method.

If the `doVerbFor()` method needs to throw one of its exceptions, it first needs to populate an exception-detail object that contains information about the exception. In particular, the method must set the status code, as Table 121 shows.

Table 121. Exception status codes for the doVerbFor() method

doVerbFor() exception	Exception status code
`ConnectionFailureException`	`APPRESPONSETIMEOUT`
`VerbProcessingFailedException`	The same outcome status code that `doVerbFor()` returns

To initialize an exception-detail object, follow these steps:

- Create the exception-detail object with the `CWConnectorExceptionObject()` constructor.
- Fill in the appropriate values of the exception-detail object with the accessor methods in the `CWConnectorExceptionObject` class, as follows:

`setMsg()`	Sets a message in the exception-detail object if there is an informational, warning, or error return message.
`setStatus()`	Sets a status return code, which is an integer whose value should be the same as shown in Table 121.

The connector framework handles copying information from the exception-detail object into the return-status descriptor that it returns to the integration broker:

- If `doVerbFor()` throws an exception, the connector framework copies the exception information.
- When `doVerbFor()` is successful, the connector framework copies the outcome status that `doVerbFor()` returns.

For more information on how to implement this method, see "Implementing the doVerbFor() method" on page 155.

See also

setErrorString(),, setStatus()

getName()

Retrieves the name of the business-object-handler object.

Syntax

```
protected String getName();
```

Parameters

None.

Return values

A `String` containing the name assigned to the business-object-handler (`CWConnectorBOHandler`) object. If `setName()` has not been called on the `CWConnectorBOHandler` object prior to this method, the method returns `null`.

Exceptions

None.

See also

setName()

setName()

Sets the name of the business-object-handler object.

Syntax

```
protected void setName(String name);
```

Parameters

name Specifies the name of the `CWConnectorBOHandler` object.

Return values

None.

Exceptions

None.

Notes

This name is typically the name of the business object the handler has been created to process.

Chapter 13. CWConnectorBusObj class

The CWConnectorBusObj class gives a view of the business object to the Java connectors developers. The class defines methods for getting information about the business object definition, business object, and its attributes. It also includes methods to obtain the metadata of the business object, and methods for reading and modifying the business object instance. Each instance of CWConnectorBusObj represents a single business object. Any manipulations of the business object has to be from this class.

Note: The CWConnectorBusObj class stores an internal handle to the BusinessObjectInterface interface of the low-level Java connector library. For more information on the classes of the low-level Java connector library, see Chapter 26, "Overview of the low-level Java connector library," on page 405.

Table 122 summarizes the methods in the CWConnectorBusObj class.

Table 122. Member methods of the CWConnectorBusObj class

Member method	Description	Page
areAllPrimaryKeysTheSame()	Determines if the attribute values in the primary key of a specified business object match those in the current business object.	261
compare()	Compares a specified business object with the current business object, based on the verb set, attribute count, application-specific information for the business object, and the attributes and attribute values.	261
doVerbFor()	Invokes the business object handler to perform the verb processing for the active verb in the business object.	262
dump()	Returns business object information in a readable format for logging and tracing.	263
getAppText()	Retrieves the value of the AppSpecificInfo field associated with this business object definition or with a specified attribute.	264
getAttrASIHashtable()	Parses the application-specific information for any attribute in a business object, given the attribute's name or its position in the business object's attribute list, into name/value pairs.	265
getAttrCount()	Retrieves the number of attributes that are in the business object's attribute list.	266
getAttrIndex()	Retrieves the ordinal position of a given attribute of a business object.	267
getAttrName()	Retrieves the name of an attribute that you specify by its position in the business object's attribute list.	267
getbooleanValue()	Retrieves the value of a boolean attribute in a business object, given the attribute's name or its position in the business object's attribute list.	268
getBusinessObjectVersion()	Retrieves the version of the business object definition.	268
getBusObjASIHashtable()	Parses the application-specific information for a business object definition into name/value pairs.	269
getBusObjValue()	Retrieves the value of an attribute that contains a business object, given the attribute's name or its position in the business object's attribute list.	269

Table 122. Member methods of the CWConnectorBusObj class (continued)

Member method	Description	Page
getCardinality()	Retrieves the cardinality of an attribute, given the attribute's name or its position in the business object's attribute list.	270
getDefault()	Retrieves the default value for this attribute, given the attribute's name or its position in the business object's attribute list.	271
getDefaultboolean()	Retrieves the default value of a `double` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	284
getDefaultdouble()	Retrieves the version of the business object definition.	272
getDefaultfloat()	Retrieves the default value of a `float` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	273
getDefaultint()	Retrieves the default value of a `int` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	274
getDefaultlong()	Retrieves the default value of a `long` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	274
getDefaultString()	Retrieves the default value of a `String` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	275
getdoubleValue()	Retrieves the value of a `double` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	276
getfloatValue()	Retrieves the value of a `float` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	276
getintValue()	Retrieves the value of a `int` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	277
getLocale()	Retrieves the locale associated with the business object.	278
getlongValue()	Retrieves the value of a `long` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	279
getLongTextValue()	Retrieves the value of a `longText` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	278
getMaxLength()	Retrieves the maximum length of an attribute from the business object definition	280
getName()	Retrieves the name of the business object definition that the current business object references.	280
getObjectCount()	Retrieves the number of child business objects in an attribute that is a business object array.	280
getParentBusinessObject()	Retrieves the parent business object of the current business object.	281
getStringValue()	Retrieves the value of a `String` attribute in a business object, given the attribute's name or its position in the business object's attribute list.	281
getSupportedVerbs()	Retrieves the supported verbs for the current business object.	282
getTypeName()	Retrieves the name of the attribute's data type, given the attribute's name or its position in the business object's attribute list.	283

Table 122. Member methods of the CWConnectorBusObj class (continued)

Member method	Description	Page
getTypeNum()	Retrieves the numeric type code for the data type of an attribute, given the attribute's name or its position in the business object's attribute list.	284
getVerb()	Retrieves the active verb for the business object.	284
getVerbAppText()	Retrieves the value of the `AppSpecificInfo` field for a particular verb.	285
hasAllKeys()	Determines if the current business object has values for *all* its primary- and foreign-key attributes.	285
hasAllPrimaryKeys()	Determines if the current business object has values for *all* its primary-key attributes.	286
hasAnyActivePrimaryKey()	Determines if the current business object has values for *any* primary-key attribute.	287
hasCardinality()	Determines if the attribute has the same cardinality as a specified cardinality value, given the attribute's name or its position in the business object's attribute list.	287
hasName()	Determines if the name of the attribute matches a specified name, given the attribute's name or its position in the business object's attribute list.	288
hasType()	Determines if the data type of the attribute matches a specified data type name.	288
isBlank()	Determines if an attribute is a part of the foreign key of the business object, given the attribute's name or its position in the business object's attribute list.	289
isForeignKeyAttr()	Determines if an attribute is a part of the foreign key of the business object, given the attribute's name or its position in the business object's attribute list.	290
isIgnore()	Determines whether the value is the special Ignore value for the attribute with the specified name or at the specified position in the attribute list.	290
isKeyAttr()	Determines if an attribute is a part of the business object primary key, given the attribute's name or its position in the business object's attribute list.	291
isMultipleCard()	Determines if an attribute has multiple cardinality, given the attribute's name or its position in the business object's attribute list.	291
isObjectType()	Determines if an attribute's data type is an object type; that is, if it is a complex attribute (an array or a subobject).	292
isRequiredAttr()	Determines if an attribute is a required attribute for the business object, given the attribute's name or its position in the business object's attribute list. If the attribute is required, it must have a value.	292
isType()	Determines if an attribute value has the same data type as a specified value.	293
isVerbSupported()	Determines whether the verb passed to the method is supported by this business object definition.	293
objectClone()	Copies an existing business object.	294
prune()	Removes the child business objects from the current (parent) business object and sets their attributes to `null`.	294
removeAllObjects()	Removes all child business objects in an attribute that is a business object array.	295
removeBusinessObjectAt()	Removes a child business object at a specified position in a business object array.	295
setAttrValues()	Sets the attributes for the current business object based on the values in a vector.	296

Table 122. Member methods of the CWConnectorBusObj class (continued)

Member method	Description	Page
setbooleanValue()	Sets the value of a `boolean` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.	296
setBusObjValue()	Sets the value of an attribute that contains a business object to a specified value, given the attribute's name or its position in the business object's attribute list.	297
setDefaultAttrValues()	Sets default values for attributes which currently have the Blank or Ignore attribute values.	299
setdoubleValue()	Sets the value of a `double` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.	299
setfloatValue()	Sets the value of a `float` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.	300
setintValue()	Sets the value of an `int` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.	301
setLocale()	Sets the locale associated with the business object.	301
setLongTextValue()	Sets the value of an `longText` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.	302
setStringValue()	Sets the value of a `String` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.	303
setVerb()	Sets the active verb for a business object.	303

As Table 122 shows, the `CWConnectorBusObj` class combines the following business object information into a single class:

- Business object definition and business object

compare()	getVerb()
`doVerbFor()`	getVerbAppText()
`dump()`	hasAnyActivePrimaryKey()
`getAppText()`	hasAllKeys()
getAttrCount()	hasAnyActivePrimaryKey()
getAttrIndex()	hasName()
getAttrName()	isVerbSupported()
getBusinessObjectVersion()	objectClone()
getBusObjASIHashtable()	prune()
getlongValue()	setAttrValues()
getName()	setVerb()
getParentBusinessObject()	

- Business object array

getObjectCount()	removeBusinessObjectAt()
removeAllObjects()	

- Business object attributes

areAllPrimaryKeysTheSame()	getTypeNum()
`getAppText()`	hasCardinality()

getAttrASIHashtable()	hasName()
getbooleanValue()	hasType()
getBusObjValue()	isBlank()
getCardinality()	isForeignKeyAttr()
getDefault()	isIgnore()
getDefaultboolean()	isKeyAttr()
getDefaultdouble()	isMultipleCard()
getDefaultfloat()	isObjectType()
getDefaultint()	isRequiredAttr()
getDefaultlong()	isType()
getdoubleValue()	setbooleanValue()
getfloatValue()	setBusObjValue()
getintValue()	setDefaultAttrValues()
getlongValue()	setdoubleValue()
getMaxLength()	setfloatValue()
getStringValue()	setintValue()
getTypeName()	setStringValue()

areAllPrimaryKeysTheSame()

Determines if the attribute values in the primary key of a specified business object match those in the current business object.

Syntax

```
public final boolean areAllPrimaryKeysTheSame(CWConnectorBusObj theBusObj);
```

Parameters

theBusObj Is the business object whose primary key values are compared to those of the current business object.

Return values

Returns `true` if all primary-key values in the *busObj* object match those in the current business object; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the attribute position specified is not valid for the definition of this business object.

`WrongAttributeException`
Thrown if the specified attribute .

See also

hasAnyActivePrimaryKey(),, hasAllKeys(), hasAllPrimaryKeys()

compare()

Compares a specified business object with the current business object, based on the verb set, attribute count, application-specific information for the business object, and the attributes and attribute values.

Syntax

```
public boolean compare(CWConnectorBusObj theBusObj);
```

Parameters

theBusObj Is the business object to compare with the current business object.

Return values

Returns `true` if all of the following information in the *busObj* object match those in the current business object:

- value of the active verb
- application-specific information for the business object definition
- attribute count
- attributes and attribute values.

For each failure, the method logs a message and returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if an attribute is not found in the definition of this business object.

`WrongAttributeException`
Thrown if the attribute types are invalid for the attributes being compared.

---

doVerbFor()

Invokes the business object handler to perform the verb processing for the active verb in the business object.

Syntax

```
public final int doVerbFor(CWConnectorReturnStatusDescriptor rtnStat);
```

Parameters

rtnStat Is an empty return-status descriptor object, which the `doVerbFor()` method populates with a status and message for the execution status of this method. The calling code can access the execution status from this return-status descriptor.

Return values

An integer that specifies the outcome status of the verb operation. Compare this integer value with the following outcome-status constants to determine the status:

`CWConnectorConstant.SUCCEED`
The verb operation succeeded.

`CWConnectorConstant.FAIL`
The verb operation failed.

`CWConnectorConstant.APPRESPONSETIMEOUT`
The application is not responding.

`CWConnectorConstant.VALCHANGE`
At least one value in the business object changed.

`CWConnectorConstant.VALDUPES`
The requested operation found multiple records for the same key value.

`CWConnectorConstant.MULTIPLE_HITS`
The connector finds multiple matching records when retrieving with non-key values. The connector will only return the first matching record in a business object.

`CWConnectorConstant.RETRIEVEBYCONTENT_FAILED`
The connector was not able to find matches for Retrieve by non-key values.

`CWConnectorConstant.BO_DOES_NOT_EXIST`
The requested business object entity does not exist in the database.

Exceptions

None.

Notes

The `doVerbFor()` method invokes the business object handler (`CWConnectorBOHandler` object) to perform the action specified by the active verb in the business object. The business object handler provides all the operations for the verbs that the business object definition supports. The active verb is one of the list of verbs that the business object definition contains. To determine the active verb for a business object, you can use the `getVerb()` method.

Within the `doVerbFor()` method, the empty passed-in *`rtnStat`* return-status descriptor is populated with a status and message to indicate the execution status of the verb processing. The calling code can then use the accessor methods of the `CWConnectorReturnStatusDescriptor` class to obtain execution information about the verb processing from the populated return-status descriptor.

This `doVerbFor()` method is normally called from the `pollForEvents()` method in the connector class (`CWConnectorAgent`) to obtain the application information for an event. The default implementation of `pollForEvents()` calls the `getBO()` method of the `CWConnectorEventStore` class to obtain application information. The `getBO()` method calls the `doVerbFor()` method in the `CWConnectorBusObj` class. If you do not use `getBO()` in your `pollForEvents()` method, you can call `doVerbFor()` directly from `pollForEvents()` by passing in an instantiated return-status descriptor. You can then obtain verb-processing status from the populated return-status descriptor once `doVerbFor()` exits.

See also

doVerbFor() (in `CWConnectorBOHandler`), getVerb(),, pollForEvents(),, setVerb()

dump()

Returns business object information in a readable format for logging and tracing.

Syntax

```
public String dump();
```

Parameters

None.

Return values

A `String` that contains the formatted business object information.

Exceptions

None.

---

getAppText()

Retrieves the value of the `AppSpecificInfo` field associated with this business object definition or with a specified attribute.

Syntax

```
public String getAppText();
public String getAppText(String attrName);
public String getAppText(int position);

public final String getAppText(String tagName, String delimiter);
public final String getAppText(String attrName, String tagName,
      String delimiter);
public final String getAppText(int position, String tagName,
      String delimiter);
```

Parameters

attrName	Is the name of an attribute whose application-specific information is parsed.
delimiter	Is the delimiter between each name-value pair. By convention, the colon (:) is used as the delimiter for building the name-value pairs.
position	Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.
tagName	Is the name of the tag in the application-specific information whose value the method retrieves.

Return values

A `String` object that holds the application-specific information from the appropriate `AppSpecificInfo` field:

- The first form of `getAppText()` retrieves application-specific information for the business object definition associated with the current business object. This method can return `null` if there is no application-specific information for the business object definition.
- The second and third forms of `getAppText()` retrieve the application-specific information for the attribute, which can be specified by name or by its position within the business object definition. This method can return `null` if there is no application-specific information for the attribute.

Exceptions

The second, third, fifth, and sixth forms of the `getAppText()` method can throw the following exception:

AttributeNotFoundException
Thrown when the specified attribute cannot be found.

The fourth, fifth, and sixth forms of the `getAppText()` method can throw the following exception:

WrongASIFormatException
Thrown if the application-specific information does not conform to the name-value format.

Notes

The `getAppText()` method provides the following forms:

- This first form retrieves the business-object-level application-specific information; that is, it obtains the application-specific information for the business object definition associated with the current business object.
- The second and third forms retrieve the attribute application-specific information; that is, they obtain the application-specific information for an attribute, which you can identify through its name (*attrName*) or position within the business object definition (*position*).
- The fourth, fifth, and sixth forms retrieve application-specific information when this information is formatted into name-value pairs of the form:

  tagName=value

  The *tagName* specifies the name of the tag (property) that appears in the application-specific information. The *delimiter* specifies the symbol that separates each name-value pair. By convention, the delimiter is usually the colon (:). The fourth form retrieves a name-value pair from the business-object-level application-specific information, while the fifth and sixth forms retrieve a name-value pair from the application-specific information of a specified attribute.

For example, suppose a business object definition contains the following application-specific information:

```
TN=table1:SCH=schema1
```

The following call to `getAppText()` retrieves the value of the name-value pair for the `TN` tag:

```
String TNvalue = busObj.getAppText("TN", ":");
```

Note: To retrieve *all* name-value pairs as a Java `Hashtable` object, use the `getBusObjASIHashtable()` or the `getAttrASIHashtable()` method for business-object-level or attribute application-specific information, respectively.

See also

getAttrASIHashtable(),, getBusObjASIHashtable(),, getVerbAppText()

getAttrASIHashtable()

Parses the application-specific information for any attribute in a business object, given the attribute's name or its position in the business object's attribute list, into name-value pairs.

Syntax

```
public final Hashtable getAttrASIHashtable(int attrName,
   String delimiter);
public final Hashtable getAttrASIHashtable(int position,
   String delimiter);
```

Parameters

attrName Is the name of an attribute whose application-specific information is parsed.

delimiter Is the delimiter between each name-value pair. Use the colon (:) as the delimiter for building the name-value pairs.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

A `java.util.Hashtable` object that contains the name-value pairs in the attribute's application-specific information.

Exceptions

`AttributeNotFoundException`
Thrown if the specified attribute cannot be found; for example, if the position specified is not valid for the definition of this business object.

`WrongASIFormatException`
Thrown if the application-specific information does not conform to the name-value format.

Notes

The `getAttrASIHashtable()` method parses the application-specific information for any attribute and returns a hash table of the name-value pairs. For example, these name/value pairs could appear as:

```
ASI=CN=colname:FK=attr1:UID=attr2:...
```

This example assumes that a colon (:) is specified as the delimiter.

Note: To retrieve *one* particular name-value pair from attribute application-specific information, use the `getAppText()` method.

See also

getAppText(),, getBusObjASIHashtable()

---

getAttrCount()

Retrieves the number of attributes that are in the business object's attribute list.

Syntax

```
public int getAttrCount();
```

Parameters

None.

Return values

An integer that specifies the number of attributes in the attribute list.

Exceptions

None.

See also

getAttrIndex()

getAttrIndex()

Retrieves the ordinal position of a given attribute of a business object.

Syntax

```
public int getAttrIndex(String attrName);
```

Parameters

attrName Is the name of the attribute in the business object definition.

Return values

The ordinal position of the attribute within the business object definition.

Exceptions

`AttributeNotFoundException`
Thrown if the attribute name specified is not valid for the definition of this business object.

getAttrName()

Retrieves the name of an attribute that you specify by its position in the business object's attribute list.

Syntax

```
public String getAttrName(int position);
```

Parameters

position Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

Return values

The name of the specified attribute.

Exceptions

`AttributeNotFoundException`
Thrown if the attribute position specified is not valid for the definition of this business object.

getbooleanValue()

Retrieves the value of a `boolean` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public boolean getbooleanValue(String attrName);
public boolean getbooleanValue(int position);
```

Parameters

attrName Is the name of an attribute whose value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The `boolean` value of the specified attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`boolean` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

See also

getAttrName(),, getBusObjValue(),, getDefaultboolean(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, getStringValue(),, setbooleanValue()

getBusinessObjectVersion()

Retrieves the version of the business object definition.

Syntax

```
public String getBusinessObjectVersion();
```

Parameters

None.

Return values

The version number of the business object.

Exceptions

None.

Notes

The version is represented by the major, minor, and point components -x.y.z. For example: - 1.0.2.

getBusObjASIHashtable()

Parses the application-specific information for a business object definition into name-value pairs.

Syntax

```
public Hashtable getBusObjASIHashtable(String delimiter);
```

Parameters

delimiter Is the delimiter between each name-value pair. Use the colon (:) as the delimiter for building the name-value pairs.

Return values

A `java.util.Hashtable` object that contains the name-value pairs in the application-specific information of the business object definition.

Exceptions

`WrongASIFormatException`
Thrown when the application-specific information does not conform to the name-value pair format.

Notes

The `getBusObjASIHashtable()` method parses the application-specific information for the business object definition associated with the current business object and returns a hash table of the name-value pairs. For example, these name-value pairs could appear as:

```
ASI=CN=colname:FK=attr1:UID=attr2:...
```

This example assumes that a colon (:) is specified as the delimiter.

Note: To retrieve *one* particular name-value pair from business-object-level application-specific information, use the `getAppText()` method.

See also

getAppText(),, getAttrASIHashtable()

getBusObjValue()

Retrieves the value of an attribute that contains a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public CWConnectorBusObj getBusObjValue(String attrName);
public CWConnectorBusObj getBusObjValue(int position);
public CWConnectorBusObj getBusObjValue(String attrName,
      int arrayIndex);
public CWConnectorBusObj getBusObjValue(int position,
      int arrayIndex);
```

Parameters

attrName Is the name of an attribute whose value is retrieved.

arrayIndex Is the integer that specifies the ordinal position of the business object within the business object array (when the attribute contains a business object array).

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The business object contained in the specified attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on an attribute that is not a business object.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

Notes

The `getBusObjValue()` method provides two forms:

- The first form expects the name or position of an attribute that is an object type. It returns the business object at the specified attribute. It assumes that the attribute has single cardinality.
- The second form expects either the name or position of an attribute and an index into a business object array. It returns the child business object at the specified index position in the business object array. It assumes that the attribute has multiple cardinality.

See also

getAttrName(),, getbooleanValue(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(),, getParentBusinessObject(),, getObjectCount(),, getStringValue(),, setBusObjValue()

---

getCardinality()

Retrieves the cardinality of an attribute, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public String getCardinality(String attrName);
public String getCardinality(int position);
```

Parameters

attrName Is the name of an attribute whose cardinality is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

A String containing the cardinality of the attribute. The value of the string is either:

`1` attribute has single cardinality or is a simple attribute

`n` attribute has multiple cardinality

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

hasCardinality(),, isMultipleCard()

getDefault()

Retrieves the default value for this attribute, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public String getDefault(String attrName);
public String getDefault(int position);
```

Parameters

attrName Is the name of an attribute whose default value is retrieved.

position Is an integer that specifies the ordinal yposition of an attribute in the business object's attribute list.

Return values

The `String` containing the default value of the attribute. If no default value exists for the attribute, the method returns an empty string.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

getDefaultboolean()

Retrieves the default value of a `boolean` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public boolean getDefaultboolean(String attrName);
public boolean getDefaultboolean(int position);
```

Parameters

attrName Is the name of an attribute whose default value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The default value of the specified attribute, as a `boolean` value, or `null` if there is no default value for the attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`boolean` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

See also

getbooleanValue(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultint(),, getDefaultlong(),, getDefaultString()

---

getDefaultdouble()

Retrieves the default value of a `double` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public double getDefaultdouble(String attrName);
public double getDefaultdouble(int position);
```

Parameters

attrName Is the name of an attribute whose default value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The default value of the specified attribute, as a `double` value, or `null` if there is no default value for the attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`double` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

`AttributeValueException`
Thrown if the default value is not in the correct format.

See also

getDefaultboolean(),, getDefaultfloat(),, getDefaultint(),, getDefaultlong(),, getDefaultString(),, getdoubleValue(),

---

getDefaultfloat()

Retrieves the default value of a `float` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public float getDefaultfloat(String attrName);
public float getDefaultfloat(int position);
```

Parameters

attrName Is the name of an attribute whose default value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The default value of the specified attribute, as a `float` value, or `null` if there is no default value for the attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`float` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

`AttributeValueException`
Thrown if the default value is not in the correct format.

See also

getDefaultboolean(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultint(),, getDefaultlong(),, getDefaultString(),, getfloatValue()

getDefaultint()

Retrieves the default value of a `int` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public int getDefaultint(String attrName);
public int getDefaultint(int position);
```

Parameters

attrName Is the name of an attribute whose default value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The default value of the specified attribute, as an `int` value, or `null` if there is no default value for the attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`int` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

`AttributeValueException`
Thrown if the default value is not in the correct format.

See also

getDefaultboolean(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultlong(),, getDefaultString(),, getintValue()

getDefaultlong()

Retrieves the default value of a `long` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public long getDefaultlong(String attrName);
public long getDefaultlong(int position);
```

Parameters

attrName Is the name of an attribute whose default value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The default value of the specified attribute, as a `long` value, or `null` if there is no default value for the attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`long` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

`AttributeValueException`
Thrown if the default value is not in the correct format.

See also

getDefaultboolean(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultlong(),, getDefaultString(),, getintValue()

---

getDefaultString()

Retrieves the default value of a `String` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public String getDefaultString(String attrName);
public String getDefaultString(int position);
```

Parameters

attrName Is the name of an attribute whose default value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The default value of the specified attribute, as a `String` value, or `null` if there is no default value for the attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`String` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

getDefaultboolean(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultint(),, getDefaultlong(),, getStringValue()

getdoubleValue()

Retrieves the value of a `double` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public double getdoubleValue(String attrName);
public double getdoubleValue(int position);
```

Parameters

attrName Is the name of an attribute whose value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The `double` value of the specified attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`double` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

`AttributeValueException`
Thrown if the `double` value is not in the correct format.

See also

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultdouble(),, getfloatValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, getStringValue(),, setdoubleValue()

getfloatValue()

Retrieves the value of a `float` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public float getfloatValue(String attrName);
public float getfloatValue(int position);
```

Parameters

attrName Is the name of an attribute whose value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The `float` value of the specified attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`float` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

`AttributeValueException`
Thrown if the `float` value is not in the correct format.

See also

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultfloat(),, getdoubleValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, getStringValue(),, setfloatValue()

---

getintValue()

Retrieves the value of a `int` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public int getintValue(String attrName);
public int getintValue(int position);
```

Parameters

attrName Is the name of an attribute whose value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The `int` value of the specified attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`int` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

`AttributeValueException`
Thrown if the `int` value is not in the correct format.

See also

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultint(),, getdoubleValue(),, getfloatValue(),, getlongValue(),, getLongTextValue(),, getStringValue(),, setintValue()

getLocale()

Retrieves the locale associated with the business object.

Syntax

```
public String getLocale();
```

Parameters

None.

Return values

The `String` that contains the name of the locale associated with the current business object.

Exceptions

None.

Notes

The `getLocale()` method returns the business-object locale, which is associated with the business object. This locale indicates the language and code encoding associated with the data in the business object, *not* with the name of the business object definition or its attributes (which must be characters in the code set associated with the U.S. English locale, `en_US`). If the business object does not have a locale associated with it, the connector framework assigns the connector-framework locale as the business-object locale.

See also

createBusObj(),, getGlobalLocale(),, setLocale()

getLongTextValue()

Retrieves the value of a `LongText` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public String getLongTextValue(String attrName);
public String getLongTextValue(int position);
```

Parameters

attrName	Is the name of an attribute whose value is retrieved.
position	Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The `String` that contains the `LongText` value of the specified attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`LongText` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultlong(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(), getStringValue(),, setLongTextValue()

---

getlongValue()

Retrieves the value of a `long` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public long getlongValue(String attrName);
public long getlongValue(int position);
```

Parameters

attrName Is the name of an attribute whose value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The `long` value of the specified attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on a non-`long` attribute.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
Thrown if the specified attribute has `null` as a value.

`AttributeValueException`
Thrown if the `long` value is not in the correct format.

See also

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultlong(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getLongTextValue(),, getStringValue()

getMaxLength()

Retrieves the maximum length of an attribute from the business object definition.

Syntax

```
public int getMaxLength(String attrName);
public int getMaxLength(int position);
```

Parameters

attrName Is the name of an attribute whose maximum length is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

An integer that specifies the maximum length, in bytes, that an attribute value can have.

Exceptions

`AttributeNotFound`
Thrown if the position or name specified is not valid for the definition of this business object.

`InvalidAttributePropertyException`
Thrown if the method is called on an object-type attribute.

getName()

Retrieves the name of the business object definition that the current business object references.

Syntax

```
public String getName();
```

Parameters

None.

Return values

The `String` object containing the name of a business object definition.

Exceptions

None.

See also

getBusinessObjectVersion()

getObjectCount()

Retrieves the number of child business objects in an attribute that is a business object array.

Syntax

```
public int getObjectCount(String attrName);
public int getObjectCount(int position);
```

Parameters

attrName Is the name of an attribute whose number of child objects is determined.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

An integer that indicates the number of child business objects in a business object array.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

getBusObjValue()

getParentBusinessObject()

Retrieves the parent business object of the current business object.

Syntax

```
public CWConnectorBusObj getParentBusinessObject();
```

Parameters

None.

Return values

The business object that contains the parent business object, or `null` if the current business object is a root and has no parent.

Exceptions

None.

See also

getBusObjValue()

getStringValue()

Retrieves the value of a `String` attribute in a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public String getStringValue(String attrName);
public String getStringValue(int position);
```

Parameters

attrName Is the name of an attribute whose value is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The `String` value of the specified attribute.

Exceptions

`WrongAttributeException`
Thrown if the method is called on an attribute that is not have an object type.

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultString(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, setStringValue()

---

getSupportedVerbs()

Retrieves the list ofverbs that the current business object supports.

Syntax

```
public String[] getSupportedVerbs();
```

Parameters

None.

Return values

An array of `String` objects, each of which contains a supported verb of the business object. Compare these String values with the following verb constants:

`CWConnectorConstant.VERB_CREATE`
The string representation for the Create verb.

`CWConnectorConstant.RETRIEVE`
The string representation for the Retrieve verb.

`CWConnectorConstant.UPDATE`
The string representation for the Update verb.

`CWConnectorConstant.DELETE`
The string representation for the Delete verb.

If your application supports other verbs, create your own verb constants to represent these verbs.

See also

getVerb(),, isVerbSupported()

getTypeName()

Retrieves the name of the attribute's data type, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public String getTypeName(String attrName);
public String getTypeName(int position);
```

Parameters

attrName Is the name of an attribute whose string value of its data type is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

A `String` that contains the name of the attribute's data type. Compare this `String` value with the attribute-type constants shown in Table 123 to determine the type.

Table 123. String attribute-type constants

Attribute data type	String attribute-type constant
Boolean	`BOOLSTRING`
Business object: multiple cardinality	`MULTIPLECARDSTRING`
Business object: single cardinality	`SINGLECARDSTRING`
	`CIPHERTEXTSTRING`
Date	`DATESTRING`
Double	`DOUBSTRING`
Float	`FLTSTRING`
Integer	`INTSTRING`
Invalid data type	`INVALID_TYPE_STRING`
Long text	`LONGTEXTSTRING`
String	`STRSTRING`

Note: The `CWConnectorAttrType` class defines the string attribute-type constants listed in Table 123.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

getTypeNum(),, hasType()

getTypeNum()

Retrieves the numeric type code for the data type of an attribute, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public int getTypeNum(String attrName);
public int getTypeNum(int position);
```

Parameters

attrName Is the name of an attribute whose numeric value of its data type is retrieved.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

An integer that specifies the data type of the attribute. Compare this integer value with the attribute-type constants shown in Table 124 to determine the type.

Table 124. Numeric attribute-type constants

Attribute data type	Numeric attribute-type constant
Boolean	BOOLEAN
	CIPHERTEXT
Date	DATE
Double	DOUBLE
Float	FLOAT
Integer	INTEGER
Invalid data type	INVALID_TYPE_NUM
Long text	LONGTEXT
Object	OBJECT
String	STRING

Note: The `CWConnectorAttrType` class defines the numeric attribute-type constants listed in Table 124.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

getTypeName(),, hasType()

getVerb()

Retrieves the active verb for the business object.

Syntax

```
public String getVerb();
```

Parameters

None.

Return values

A `String` object that contains the active verb for the business object. If there is no active verb for the business object, the returned String is empty.

Exceptions

None.

Notes

The business object definition contains the list of verbs that the business object supports. The `getVerb()` method enables you to determine the active verb for the current business object.

See also

isVerbSupported(),, setVerb()

getVerbAppText()

Retrieves the value of the `AppSpecificInfo` field for a particular verb.

Syntax

```
public String getVerbAppText(String verb);
```

Parameters

verb Is the verb for which the value of the `AppSpecificInfo` field is to be retrieved.

Return values

A `String` object that holds the application-specific information for the verb. This information is stored in `AppSpecificInfo` field for the specified verb. If the business object does not have application-specific information for the verb, the method returns an empty string.

Exceptions

None.

See also

getAppText(),, getVerb()

hasAllKeys()

Determines if the current business object has values for *all* its primary- and foreign-key attributes.

Syntax

```
public final boolean hasAllKeys();
```

Parameters

None.

Return values

Returns `true` if the current business object has values for *all* primary and foreign key attributes; otherwise returns `false`.

Exceptions

`WrongAttributeException`
Thrown if the key is set on a multiple cardinality attribute.

`AttributeNotFoundException`
Thrown if a key attribute cannot be found within the business object definition.

Notes

The `hasAllKeys()` method checks if all the primary and foreign keys have been populated. This method is typically used to identify the row for updates.

See also

areAllPrimaryKeysTheSame(),, hasAnyActivePrimaryKey(),, hasAllPrimaryKeys()

---

hasAllPrimaryKeys()

Determines if the current business object has values for *all* its primary-key attributes.

Syntax

```
public final boolean hasAllPrimaryKeys();
```

Parameters

None.

Return values

Returns `true` if the current business object has values for *all* primary key attributes; otherwise returns `false`.

Exceptions

`WrongAttributeException`
Thrown if the key is set on a multiple cardinality attribute.

`AttributeNotFoundException`
Thrown if a primary key attribute cannot be found within the business object definition.

Notes

The `hasAllPrimaryKeys()` method checks if all the primary keys have been populated. This method is typically used to identify the row for updates.

See also

areAllPrimaryKeysTheSame(),, hasAnyActivePrimaryKey(),, hasAllKeys()

hasAnyActivePrimaryKey()

Determines if the current business object has values for *any* primary-key attribute.

Syntax

```
public final boolean hasAnyActivePrimaryKey();
```

Parameters

None.

Return values

Returns `true` if the current business object has a value for *any* primary key attribute; otherwise returns `false`.

Exceptions

`WrongAttributeException`
: Thrown if the key is set on a multiple cardinality attribute.

`AttributeNotFoundException`
: Thrown if a key attribute cannot be found within the business object definition.

Notes

The `hasAnyActivePrimaryKey()` method checks if at least one primary key has been populated. This method is typically used to identify the row for deletes.

See also

areAllPrimaryKeysTheSame(),, hasAllKeys(),, hasAllPrimaryKeys()

hasCardinality()

Determines if the attribute has the same cardinality as a specified cardinality value, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public boolean hasCardinality(String attrName, String card);
public boolean hasCardinality(int position, String card);
```

Parameters

attrName
: Is the name of an attribute whose cardinality is tested.

card
: Is the cardinality value to use for checking. Valid cardinality values are:

```
1 - single cardinality
n - multiple cardinality
```

position
: Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the cardinality of the attribute matches the specified value; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

Notes

This method is used to test cardinality of complex attributes (subobjects and arrays).

See also

getCardinality(),, isMultipleCard()

hasName()

Determines if the name of the attribute matches a specified name, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public boolean hasName(int position, String name);
```

Parameters

name Is the name of the attribute to test for at the specified attribute position.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute name matches the specified name; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

hasType()

Determines if the data type of the attribute matches a specified data type name.

Syntax

```
public boolean hasType(String attrName, int typeName);
public boolean hasType(int position, String typeName);

public boolean hasType(String attrName, int typeNum);
public boolean hasType(int position, String typeNum);
```

Parameters

attrName Is the name of an attribute whose cardinality is tested.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

typeName Is the string value of the data type of the attribute to test for. Use one of the string attribute-type constants in Table 123 to specify the data type.

typeNum Is the numeric value of the data type of the attribute to test for. Use one of the numeric attribute-type constants in Table 124 to specify the data type.

Return values

Returns `true` if the attribute type matches the passed-in type name; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

getTypeName(),, getTypeNum(),, hasName()

---

isBlank()

Determines whether the value is the special Blank attribute value for the attribute with the specified name or at the specified position in the attribute list.

Syntax

```
public boolean isBlank(String attrName);
public boolean isBlank(int position);
```

Parameters

attrName Is the name of an attribute whose value is checked for blank.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute value equals the blank value or `false` if it does not.

Exceptions

None.

See also

isIgnore()

isForeignKeyAttr()

Determines if an attribute is a part of the foreign key of the business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public boolean isForeignKeyAttr(String attrName);
public boolean isForeignKeyAttr(int position);
```

Parameters

attrName Is the name of an attribute that is checked for participation in a foreign key.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute is a foreign key, or part of the foreign key, for the business object; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

hasAllKeys(),, isKeyAttr()

isIgnore()

Determines whether the value is the special Ignore value for the attribute with the specified name or at the specified position in the attribute list.

Syntax

```
public boolean isIgnore(String attrName);
public boolean isIgnore(int position);
```

Parameters

attrName Is the name of an attribute whose value is checked for "ignore".

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute value equals the special "ignore" value or `false` if it does not.

Exceptions

None.

See also

isBlank()

isKeyAttr()

Determines if an attribute is a part of the business object primary key, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public boolean isKeyAttr(String attrName);
public boolean isKeyAttr(int position);
```

Parameters

attrName Is the name of an attribute that is checked for participation in a key.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute is a primary key, or part of the primary key, for the business object; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

areAllPrimaryKeysTheSame(),, hasAnyActivePrimaryKey(),, hasAllKeys(),, hasAllPrimaryKeys(),, isForeignKeyAttr()

isMultipleCard()

Determines if an attribute has multiple cardinality, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public boolean isMultipleCard(String attrName);
public boolean isMultipleCard(int position);
```

Parameters

attrName Is the name of an attribute that is checked for multiple cardinality.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute is a multiple cardinality; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

See also

getCardinality(),, hasCardinality()

isObjectType()

Determines if an attribute's data type is an object type; that is, if it is a complex attribute (an array or a subobject).

Syntax

```
public boolean isObjectType(String attrName);
public boolean isObjectType(int position);
```

Parameters

attrName Is the name of an attribute that is checked for an object data type.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute is a business object or a complex attribute, such as a business object array or subobject; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

isRequiredAttr()

Determines if an attribute is a required attribute for the business object, given the attribute's name or its position in the business object's attribute list. If the attribute is required, it must have a value.

Syntax

```
public boolean isRequiredAttr(String attrName);
public boolean isRequiredAttr(int position);
```

Parameters

attrName Is the name of an attribute that is checked to see if it is required.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute is required for the business object; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

isType()

Determines if an attribute value has the same data type as a specified value.

Syntax

```
public boolean isType(String attrName, Object value);
public boolean isType(int position, Object value);
```

Parameters

attrName Is the name of an attribute whose data type is compared with the specified attribute value.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

value Is the value whose data type is compared with the attribute value.

Return values

Returns `true` if the type of the attribute matches the passed-in type; otherwise, returns `false`.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

isVerbSupported()

Determines whether the verb passed to the method is supported by this business object definition.

Syntax

```
public boolean isVerbSupported(String verb);
```

Parameters

verb Is the verb that the method determines if the current business object definition supports.

Return values

Returns `true` if the specified verb is supported; otherwise, returns `false`.

Exceptions

None.

See also

getVerb(),, getSupportedVerbs()

objectClone()

Copies an existing business object.

Syntax

```
public CWConnectorBusObj objectClone();
```

Parameters

None.

Return values

A copy of the current business object, including its attributes and verbs.

Exceptions

None.

Notes

This method copies the business object attributes and also its verb.

prune()

Removes the child business objects from the current (parent) business object and sets their attributes to `null`.

Syntax

```
public final void prune();
```

Parameters

None.

Return values

None.

Exceptions

`AttributeNotFoundException`
: Thrown if the object-type attribute is not found in the definition of this business object.

`WrongAttributeException`
: Thrown if the attribute is not valid (not an object-type attribute).

removeAllObjects()

Removes all child business objects in an attribute that is a business object array.

Syntax

```
public void removeAllObjects(String attrName);
public void removeAllObjects(int position);
```

Parameters

attrName Is the name of an attribute whose business objects are removed from its business object array.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

removeBusinessObjectAt()

Removes a child business object at a specified position in a business object array.

Syntax

```
public void removeBusinessObjectAt(String attrName, int index);
public void removeBusinessObjectAt(int position, int index);
```

Parameters

attrName Is the name of an attribute whose business objects are removed from its business object array.

index Is an integer that specifies the position for a child business object in a business object array.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

Notes

After the remove operation, the business object array is compacted. Indexes are decremented for all business objects that have an index number higher than that of the removed business object.

setAttrValues()

Sets the attributes for the current business object based on the values in a vector.

Syntax

```
public final void setAttrValues(Vector attrValues);
```

Parameters

attrValues Is a `java.util.Vector` object that contains a value for each attribute in the current business object.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if a value specified in the *attrValues* vector does not have an associated attribute in the definition of this business object.

`AttributeValueException`
Thrown if the attribute value in the *attrValues* vector is not compatible with its associated attribute's data type.

`WrongAttributeException`
Thrown if the value is being set on an object-type attribute.

setbooleanValue()

Sets the value of a `boolean` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public void setbooleanValue(String attrName, boolean newVal);
public void setbooleanValue(int position, boolean newVal);
```

Parameters

attrName Is the name of the attribute whose value you want to set.

position Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

newVal Is the `boolean` value to assign to the attribute.

Return values

None.

Exceptions

`AttributeNotFoundException`
: Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeValueException`
: Thrown if the value passed in is not a valid value for the particular attribute.

`WrongAttributeException`
: Thrown if the value is being set on a non-`boolean` attribute.

See also

getbooleanValue(),, getDefaultboolean(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

---

setBusObjValue()

Sets the value of an attribute that contains a business object to a specified value, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public void setBusObjValue(String attrName, CWConnectorBusObj newVal);
public void setBusObjValue(int position, CWConnectorBusObj newVal);
public void setBusObjValue(String attrName, CWConnectorBusObj newVal,
      int arrayIndex);
public void setBusObjValue(int position, CWConnectorBusObj newVal,
      int arrayIndex);
```

Parameters

attrName
: Is the name of the attribute whose value you want to set.

arrayIndex
: Is the integer that specifies the ordinal position of the business object within the business object array (when the attribute contains a business object array).

position
: Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

newVal
: Is the `boolean` value to assign to the attribute.

Return values

None.

Exceptions

`AttributeNotFoundException`
: Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeNullValueException`
: Thrown if the business object array to hold the business object could not be created (for a multiple cardinality attribute).

`WrongAttributeException`
: Thrown if the value is being set on a non-object attribute.

`AttributeValueException`
Thrown if the value to be set is not a valid business object.

`SpecNameNotFoundException`
Thrown if the business object definition for the business object array could not be found. This exception is returned only by the forms of `setBusObjValue()` that pass in the *`arrayIndex`* argument.

Notes

The `setBusObjValue()` method provides two forms:

- The first form expects the name or position of an attribute that is an object type and the business object to assign to this attribute. It assumes that the attribute has single cardinality.
- The second form expects:
  - the name or position of the attribute to set
  - the business object to assign to the attribute
  - an index position within the business object array at which to assign the object value

  It assumes that the attribute has multiple cardinality.

See also

getBusObjValue(),, setbooleanValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

---

setDEEId()

Sets the ObjectEventId attribute to a specified event identifier (ID).

Syntax

```
public void setDEEId(String eventId);
```

Parameters

eventId Is the event identifier you want to assign to the ObjectEventId attribute.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeValueException`
Thrown if the value to be set is not a valid business object.

Notes

In the duplication event elimination feature, the business object must store the event ID for its event record in its ObjectEventId attribute. Normally, the ObjectEventId is reserved for use by the integration broker. To access this attribute

for the duplication event elimination feature, use the `setDEEId()` method. For more information, see the description of duplicate event elimination in Chapter 5, "Event notification," on page 115.

setDefaultAttrValues()

Sets default values for attributes which currently have the Blank or Ignore values.

Syntax

```
public void setDefaultAttrValues();
```

Parameters

None.

Return values

None.

Exceptions

None.

Notes

The `setDefaultAttrValues()` method sets default values as valid values, not Ignore values. For complex attributes (whose type a business object or business object array), the method creates an empty container. The method sets default values for instances of subobjects within the business object.

See also

setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

setdoubleValue()

Sets the value of a `double` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public void setdoubleValue(String attrName, double newVal);
public void setdoubleValue(int position, double newVal);
```

Parameters

attrName Is the name of the attribute whose value you want to set.

position Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

newVal Is the `double` value to assign to the attribute.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeValueException`
Thrown if the value passed in is not a valid value for the particular attribute.

`WrongAttributeException`
Thrown if the value is being set on a non-`double` attribute.

See also

getDefaultdouble(),, getdoubleValue(),, setbooleanValue(),, setBusObjValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

---

setfloatValue()

Sets the value of a `float` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public void setfloatValue(String attrName, float newVal);
public void setfloatValue(int position, float newVal);
```

Parameters

attrName Is the name of the attribute whose value you want to set.

position Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

newVal Is the `float` value to assign to the attribute.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeValueException`
Thrown if the value passed in is not a valid value for the particular attribute.

`WrongAttributeException`
Thrown if the value is being set on a non-`float` attribute.

See also

getDefaultfloat(),, getfloatValue(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setintValue(),, setLongTextValue(),, setStringValue()

setintValue()

Sets the value of an `int` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public void setintValue(String attrName, int newVal);
public void setintValue(int position, int newVal);
```

Parameters

attrName Is the name of the attribute whose value you want to set.

position Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

newVal Is the `int` value to assign to the attribute.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeValueException`
Thrown if the value passed in is not a valid value for the particular attribute.

`WrongAttributeException`
Thrown if the value is being set on a non-`integer` attribute

See also

getDefaultint(),, getintValue(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setLongTextValue(),, setStringValue()

setLocale()

Sets the locale for the business object.

Syntax

```
public void setLocale(String localeName);
```

Parameters

localeName Is the name of the locale to associate with the current business object.

Return values

None.

Exceptions

`IllegalLocaleException`
Thrown if the locale name specified is not valid.

Notes

The `setLocale()` method sets the business-object locale, which identifies the locale that is associated with the business object. This locale indicates the language and code encoding associated with the data in the business object, *not* with the name of the business object definition or its attributes (which must be characters in the code set associated with the U.S. English locale, `en_US`). If the business object does not have a locale associated with it, the connector framework assigns the connector-framework locale as the business-object locale.

See also

getLocale()

---

setLongTextValue()

Sets the value of an `LongText` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public void setLongTextValue(String attrName, String newVal);
public void setLongTextValue(int position, String newVal);
```

Parameters

attrName	Is the name of the attribute whose value you want to set.
position	Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.
newVal	Is a `String` that contains the `LongText` value to assign to the attribute.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeValueException`
Thrown if the value passed in is not a valid value for the particular attribute.

`WrongAttributeException`
Thrown if the value is being set on a non-`LongText` attribute

See also

getLongTextValue(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setStringValue()

setStringValue()

Sets the value of a `String` attribute to a specified value, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public void setStringValue(String attrName, String newVal);
public void setStringValue(int position, String newVal);
```

Parameters

attrName Is the name of the attribute whose value you want to set.

position Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

newVal Is the `String` value to assign to the attribute.

Return values

None.

Exceptions

`AttributeNotFoundException`
Thrown if the position or name specified is not valid for the definition of this business object.

`AttributeValueException`
Thrown if the value passed in is not a valid value for the particular attribute.

`WrongAttributeException`
Thrown if the value is being set on a non-`String` attribute.

See also

getDefaultString(),, getStringValue(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue()

setVerb()

Sets the active verb for a business object.

Syntax

```
public void setVerb(String newVerb);
```

Parameters

newVerb Is a verb that is in the verb list of the business object definition to which the business object refers.

Return values

None.

Exceptions

`InvalidVerbException`
: Thrown if the verb passed in is not a supported verb in the business object definition.

Notes

The business object definition contains the list of verbs that the business object supports. The verb that you set as the active verb must be on this list. Only one verb is active at a time for a business object.

All business objects typically support the Create, Retrieve, and Update verbs. A business object might support additional verbs, such as Delete. Every connector that supports the business object must implement all the verbs that it supports.

See also

getVerb()

Chapter 14. CWConnectorConstant class

The CWConnectorConstant class defines the constants shared by all Java connectors. The CWConnectorConstant class provides the following groups of static constants:

- "Outcome-status constants"
- "Verb constants"
- "Connector-property constants" on page 306

Note: The CWConnectorConstant class extends the CxStatusConstants class of the low-level Java connector library. For more information on the classes of the low-level Java connector library, see Chapter 26, "Overview of the low-level Java connector library," on page 405.

Outcome-status constants

Many methods of the Java connector library return an integer outcome status to indicate the success of the method. Table 125 summarizes the static outcome-status constants, which are defined in the CWConnectorConstant class.

Table 125. Outcome-status constants of the CWConnectorConstant class

Constant name	Meaning
SUCCEED	The operation completed successfully.
APPRESPONSETIMEOUT	The application is not responding.
BO_DOES_NOT_EXIST	The requested business object in a retrieve does not exist.
CONNECTOR_NOT_ACTIVE	The connector has attempted to deliver an event but the connector controller is not active; it has been paused. Only when the integration broker is InterChange Server does a connector controller exist.
FAIL	The operation failed for an unspecified reason.
MULTIPLE_HITS	The integration broker requested a retrieve-by-content but the connector found more than one matching record. The status indicates that more than one record matched the search requirements.
NO_SUBSCRIPTION_FOUND	No subscriptions for the event.
RETRIEVEBYCONTENT_FAILED	Retrieve by content failed.
UNABLETOLOGIN	The connector cannot log into the application.
VALCHANGE	The operation successfully completed and changed the value of the object in the target application.
VALDUPES	The requested operation was not needed because the object in the application already had the requested characteristics.

Verb constants

When the doVerbFor() method of a Java connector needs to refer to one of the basic verb values, it can use the verb constants that the CWConnectorConstant class defines. Table 125 summarizes the static verb constants.

Table 126. Verb constants of the CWConnectorConstant class

Constant name	Meaning
VERB_CREATE	String representation of the Create verb
VERB_RETRIEVE	String representation of the Retrieve verb
VERB_UPDATE	String representation of the Update verb
VERB_DELETE	String representation of the Delete verb

Table 126. Verb constants of the CWConnectorConstant class (continued)

Constant name	Meaning
VERB_EXISTS	String representation of the Exists verb
VERB_RETRIEVEBYCONTENT	String representation of the RetrieveByContent verb

Verb constants are useful in the doVerbFor() method.

Connector-property constants

Many methods of the Java connector library return an integer outcome status to indicate the success of the method. Table 125 summarizes the static outcome-status constants, which are defined in the CWConnectorConstant class.

Table 127. Connector-property constants of the CWConnectorConstant class

Constant name	Meaning
HIERARCHICAL	The connector property is hierarchical; that is, it contains a combination of multiple string values and child properties.
SIMPLE	The connector property is simple; that is, it contains only string values, no child properties.
SINGLE_VALUED	The connector property contains only a single value.
MULTI_VALUED	The connector property contains one or more values.

Chapter 15. CWConnectorEvent class

The CWConnectorEvent class allows you to create and interact with connector event objects. An event object represents the occurred event in the application. The event store builds these event objects for each event pulled from the application. The information in each event object is then used to build and retrieve the business object for further processing by the connector infrastructure.

Table 128 summarizes the methods in the CWConnectorEvent class.

Table 128. Member methods of the CWConnectorEvent class

Member method	Description	Page
CWConnectorEvent()	Creates a new event object.	307
getBusObjName()	Retrieves the name of the business object associated with the event object.	308
getConnectorID()	Retrieves the connector identifier (ID) from the event object.	309
getEffectiveDate()	Retrieves the effective date from the event object.	309
getEventID()	Retrieves the event identifier (ID) from the event object.	310
getEventSource()	Retrieves the name of the event source from the event object.	310
getEventTimeStamp()	Retrieves the event timestamp from the event object.	310
getIDValues()	Retrieves the data values of the business object from the event object.	311
getKeyDelimiter()	Retrieves the key delimiter from the event object.	311
getPriority()	Retrieves the priority from the event object.	312
getStatus()	Retrieves the status from the event object.	312
getTriggeringUser()	Retrieves the triggering user from the event object.	313
getVerb()	Retrieves the verb from the event object.	313
setEventSource()	Sets the event source to a specified value in the event object.	314

CWConnectorEvent()

Creates a new event object.

Syntax

```
public CWConnectorEvent();

public CWConnectorEvent(String eventID, String busObjName,
    String verb, String IDvalues, int status, int priority,
    String connectorID, Date eventTimeStamp, Date effectiveDate,
    String triggeringUser, String description, String delimiter);
```

Parameters

busObjName Is the business object associated with the event.

connectorID Is the connector identifier (ID) for the connector associated with the event.

description Is an optional description of the event.

delimiter Is the delimiter that separates the key values of the event.

effectiveDate Is the effective date for the event.

eventID Specifies the event identifier for the event.

eventTimeStamp
Is the timestamp for the event.

IDvalues Is the data for the business object associated with the event.

priority Is an integer event priority

status Is one of the following event-status constants to associate with the event:

```
CWConnectorEventStatus.IN_PROGRESS
CWConnectorEventStatus.READY_FOR_POLL
CWConnectorEventStatus.SUCCESS
CWConnectorEventStatus.UNSUBSCRIBED
CWConnectorEventStatus.ERROR_OBJECT_NOT_FOUND
CWConnectorEventStatus.ERROR_POSTING_EVENT
CWConnectorEventStatus.ERROR_PROCESSING_EVENT
```

triggeringUser Is the user identifier (ID) associated with the user that triggered the event.

verb Is the verb for the *busObjName* business object.

Return values

A `CWConnectorEvent` object containing the newly created event.

Notes

The `CWConnectorEvent()` constructor has two forms:

- The first form creates an empty event object.
- The second form passes data to initialize the new event object. The second form of the `CWConnectorEvent()` constructor provides a way to initialize the members of the event object.

Note: The only way to initialize the event's description is through the second form of the `CWConnectorEvent()` constructor. There is no accessor method for this member because connectors do not use the event description.

---

getBusObjName()

Retrieves the name of the business object associated with the event object.

Syntax

```
public String getBusObjName();
```

Parameters

None.

Return values

A `String` object containing the name of the business object.

Exceptions

`AttributeNullValueException`
Thrown if the business object name is null.

Notes

An event store might not persist the name of the business object. In some cases, the business object name might be determined when it is created based on content.

getConnectorID()

Retrieves the connector identifier (ID) from the event object.

Syntax

```
public String getConnectorID();
```

Parameters

None.

Return values

A `String` containing the connector ID, which identifies the connector to which the event is assigned.

Exceptions

`AttributeNullValueException`
Thrown if the connector ID is null.

Notes

Currently, the connector ID is only used for tracing purposes.

getEffectiveDate()

Retrieves the effective date from the event object.

Syntax

```
public Date getEffectiveDate();
```

Return values

A `Date` object containing the event's effective date, which is the date on which the event becomes active and should be processed.

Exceptions

`AttributeNullValueException`
Thrown if the event's effective date is null.

Notes

An effective date is useful when your event detection mechanism handles future-event processing; that is, it stores events that must be processed at some particular point in the future. The effective date indicates when the event should be processed.

getEventID()

Retrieves the event identifier (ID) from the event object.

Syntax

```
public String getEventID();
```

Parameters

None.

Return values

A `String` object containing the event ID, which uniquely identifies the event.

Exceptions

`AttributeNullValueException`
Thrown if the event ID is null.

Notes

If the event store is an event table in a database, the event ID is the key value of the table row. For other event stores, the event ID can be a file name and the position of the record within the file.

getEventSource()

Retrieves the name of the event source from the event object.

Syntax

```
public String getEventSource();
```

Return values

A `String` object containing the event source, which is the source from which the event originated.

Exceptions

`AttributeNullValueException`
Thrown if the evemt source is null.

Notes

The event source is often used by connectors that require this information for archiving. For example, the WebSphere Business Integration Adapter for JText stores the name of the WebSphere MQ queue.

getEventTimeStamp()

Retrieves the event timestamp from the event object.

Syntax

```
public Date getEventTimeStamp();
```

Parameters

None.

Return values

A `String` object containing the event timestamp, which is the time the event was created.

Exceptions

`AttributeNullValueException`
Thrown if the event timestamp is null.

getIDValues()

Retrieves the data values of the business object from the event object.

Syntax

```
public String getIDValues();
```

Parameters

None.

Return values

A `String` object containing the business object's data values, which identify the business object.

Exceptions

`AttributeNullValueException`
Thrown if the data of the business object is null.

Notes

As a standard, these data values should be the key values for the business object; that is, data values in name/value pair format. They should include whatever attribute values are needed to uniquely identify a business object to be retrieved during polling.

getKeyDelimiter()

Retrieves the key delimiter from the event object.

Syntax

```
public String getKeyDelimiter();
```

Parameters

None.

Return values

A `String` object containing the event's key delimiter.

Exceptions

`AttributeNullValueException`
Thrown if the key delimiter is null.

getPriority()

Retrieves the priority from the event object.

Syntax

```
public int getPriority();
```

Parameters

None.

Return values

An integer to indicate the priority of the event.

Exceptions

None.

Notes

Use the event priority to determine the correct processing order of the event.

getStatus()

Retrieves the status from the event object.

Syntax

```
public int getStatus();
```

Parameters

None.

Return values

An integer value that represents the event status. Compare this integer value with the following event-status constants to determine the status:

```
CWConnectorEventStatus.IN_PROGRESS
CWConnectorEventStatus.READY_FOR_POLL
CWConnectorEventStatus.SUCCESS
CWConnectorEventStatus.UNSUBSCRIBED
CWConnectorEventStatus.ERROR_OBJECT_NOT_FOUND
CWConnectorEventStatus.ERROR_POSTING_EVENT
CWConnectorEventStatus.ERROR_PROCESSING_EVENT
```

Exceptions

None.

Notes

The Java connector library provides the `getStatus()` method as a public method in the `CWConnectorEvent` class. However, it does *not* provide a public method for

setting this status. To set the event status, use one of the following Java connector library methods from the CWConnectorEventStore class:

- getNextEvent()
- recoverInProgressEvents()
- resubmitArchivedEvents()
- setEventStatus()
- updateEventStatus()

getTriggeringUser()

Retrieves the triggering user from the event object.

Syntax

```
public String getTriggeringUser();
```

Parameters

None.

Return values

A String object containing the triggering user for the event, which is the user ID that triggering the event.

Exceptions

AttributeNullValueException
: Thrown if the name of the triggering user is null.

Notes

You can use the triggering user value to avoid ping pong in a standard way when synchronizing between two systems.

getVerb()

Retrieves the verb from the event object.

Syntax

```
public String getVerb();
```

Parameters

None.

Return values

A String object containing the verb associated with the event.

Exceptions

AttributeNullValueException
: Thrown if the verb is null.

setEventSource()

Sets the event source to a specified value in the event object.

Syntax

```
public void setEventSource(String eventSource);
```

Parameters

eventSource Specifies the new event source to assign to the event.

Return values

None.

Exceptions

None.

Chapter 16. CWConnectorEventStatusConstants class

The CWConnectorEventStatusConstants class defines static constants for status values that an event can have.

Event-status constants

The event-status constants are typically used in the poll method to track the current status of an event. Table 129 summarizes the static event-status constants in the CWConnectorEventStatusConstants class.

Table 129. Static constants of the CWConnectorEventStatusConstants class

Event-status constant	Meaning
ERROR_OBJECT_NOT_FOUND	Error in finding the event in the application database
ERROR_POSTING_EVENT	Error in sending the event to InterChange Server. A description of the error can be appended to the event description in the event record.
ERROR_PROCESSING_EVENT	Error in processing the event. A description of the error can be appended to the event description in the event record.
IN_PROGRESS	Event is in progress
READY_FOR_POLL	Ready for poll
SUCCESS	Sent to connector framework
UNSUBSCRIBED	No subscriptions for event

Figure 76 shows when the different event-status constants are set.

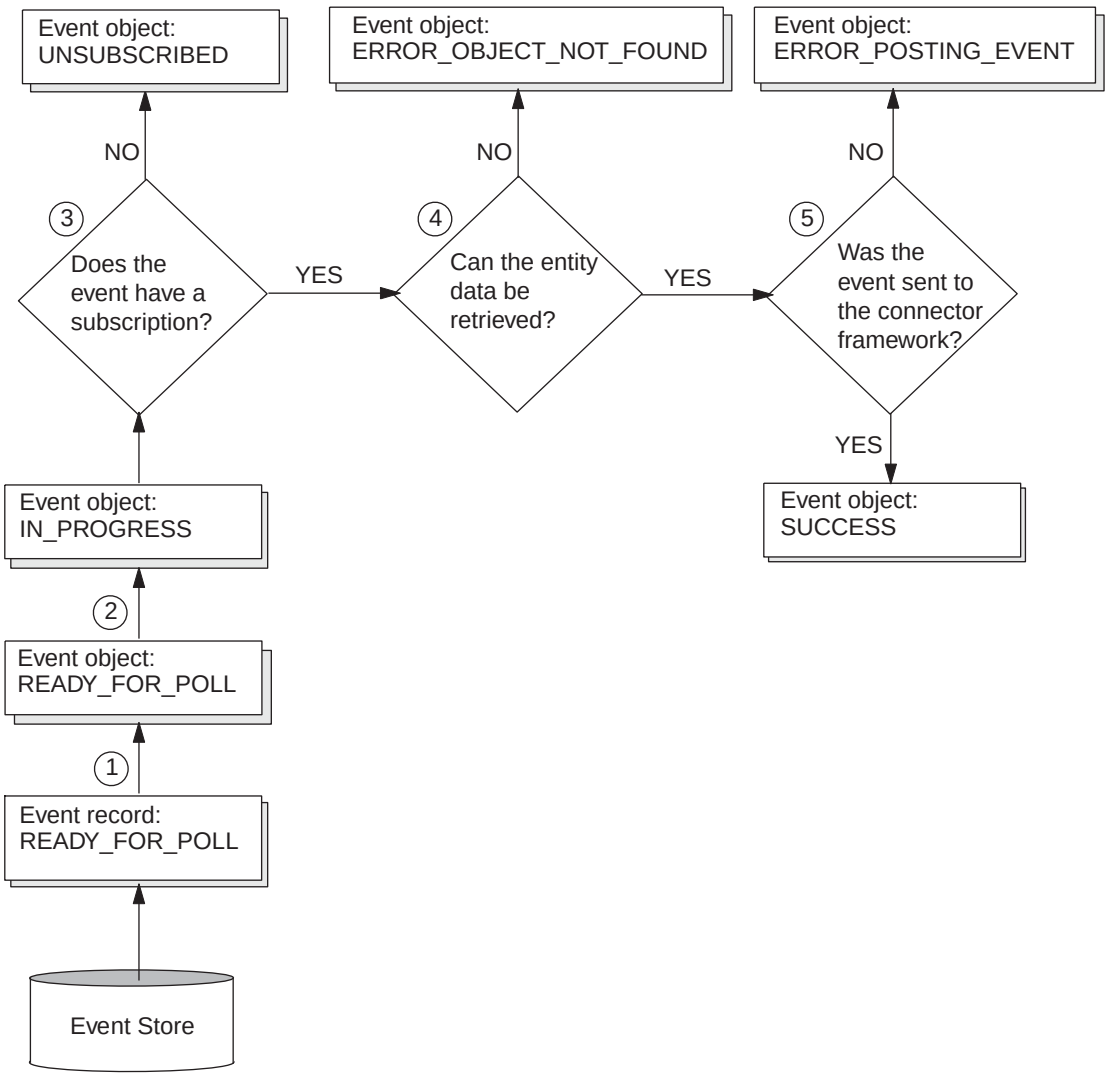

Figure 76. Event-status values for the poll method

As Figure 76 shows, the poll method takes the following steps to maintain the status of an event object:

1. The `fetchEvents()` retrieves the Ready-for-Poll event record and creates an event object with the `READY_FOR_POLL` status.
2. The `getNextEvent()` method retrieves a Ready-for-Poll event object from the events vector and update its status to `IN_PROGRESS`.
3. The poll method uses the `isSubscribed()` method to check whether the retrieved event has any subscriptions.
   - If no subscriptions exist, the poll method uses `updateEventStatus()` to change the event object's status to `UNSUBSCRIBED`.
   - If subscriptions do exist, execution of the poll event continues with step 4.
4. The poll method calls the `getBO()` method to retrieve the application entity's data to populate the business object.
   - If `getBO()` cannot locate the application entity's data, the poll method uses `updateEventStatus()` to change the event object's status to `ERROR_OBJECT_NOT_FOUND`.

- If the application entity data is found, execution of the poll event continues with step 5.

5. The poll method calls the `gotApplEvent()` method to send the business object to the connector framework, where it is then routed to its destination. The poll method uses the `updateEventStatus()` method to change the event object's status to reflect the success of `gotApplEvent()`. For a list of event status values that correspond to the `gotApplEvent()` return codes, see Table 100 on page 193.

Chapter 17. CWConnectorEventStore class

The CWConnectorEventStore class is a base class to provide a Java connector with the ability to access an event store. An event store is the application's mechanism for persistently storing events. The application stores event records in the event store for events that occur in the application. The connector retrieves events from the event store and processes them for transferal to the integration broker. From this class, a connector developer must derive an event-store class and implement some of its methods for the event store.

Important: All Java connectors *must* extend this class to access the application's event store. To access the application's event store through the Java CWConnectorEventStore class, developers must implement the following abstract methods in their derived event-store class: deleteEvent(), fetchEvents(), recoverInProgressEvents(), resubmitArchivedEvents(), and setEventStatus(). To access an archive store, developers must implement the archiveEvent() method.

Table 130 summarizes the methods in the CWConnectorEventStore class.

Table 130. Member methods of the CWConnectorEventStore class

Member method	Description	Page
CWConnectorEventStore()	Creates an event-store object	319
archiveEvent()	Archives the specified event in the application's archive store with appropriate status.	320
cleanupResources()	Releases resources that the poll method has used to access the event store.	321
deleteEvent()	Deletes the event from the application's event store.	321
fetchEvents()	Retrieves a specified number of Ready-for-Poll events from the application's event store.	322
getBO()	Builds a business object based on the information in an event from the event store.	323
getNextEvent()	Retrieves the next event object from the eventsToProcess vector.	325
recoverInProgressEvents()	Recovers any In-Progress events in the event store.	326
resubmitArchivedEvents()	Copies the specified archived event from the application's archive store to the application's event store and changes the event status to READY_FOR_POLL.	327
setEventStatus()	Sets the status of an event in the event store.	328
setEventsToProcess()		329
setTerminate()	Sets the internal terminate-connector flag to true.	329
updateEventStatus()	Updates the event status both in the application's event store and in the event.	330

CWConnectorEventStore()

Creates an event-store object.

Syntax

```
public CWConnectorEventStore();
```

Parameters

None.

Return values

A `CWConnectorEventStore` object containing the newly created event store.

Notes

The `CWConnectorEventStore()` constructor creates a new event store and initializes the single data member, `eventsToProcess`. The `eventsToProcess` member is a Java `Vector` object to hold retrieved event objects.

---

archiveEvent()

Archives the specified event in the application's archive store with appropriate status.

Syntax

```
public int archiveEvent(String eventID);
```

Parameters

eventID Specifies the event ID of the event to archive.

Return values

An integer that indicates the outcome status of the archive operation. Compare this integer value with the following outcome-status constants to determine the status:

`CWConnectorConstant.SUCCEED`
The archiving of the event succeeded.

`CWConnectorConstant.FAIL`
The archiving of the event failed.

Exceptions

`ArchiveFailedException`
Thrown when the underlying application is unable to archive the event.

`InvalidStatusChangeException`
Thrown if the connector tries to update the event status with one that is invalid for the application.

Notes

The `arhiveEvent()` method is usually called from the poll method, `pollForEvents()` to archive processed or unsuccessful events to the event archive store.

Important: The `archiveEvent()` method is *not* an abstract method because it is a synchronized method. However, the event-store class *must* implement this method to provide the ability to archive an event to the archive store.

See also

deleteEvent(),, pollForEvents()

cleanupResources()

Release resourses that the polling method has used to access the event store.

Syntax

```
public void cleanupResources();
```

Parameters

None.

Return values

None.

Exceptions

None.

Notes

The `cleanupResources()` method is useful as one of the last steps in the `pollForEvents()` method. In it, you can include code that releases resources that the `pollForEvents()` method has allocated to access the event store. For example, if the event store is implemented as an event table, the `pollForEvents()` method might have allocated SQL cursors to access the event tables. In this case, you can include statements in `cleanupResources()` that close these cursors, thereby freeing memory usage and releasing unneeded cursors.

Important: The `cleanupResources()` method is *not* an abstract method. However, neither does it provide a default implementation. Therefore, to provide the ability to clean up resources used to access your event store, you must override the default `cleanupResources()` with your own implementation.

See also

pollForEvents()

deleteEvent()

Deletes the event from the application's event store.

Syntax

```
public abstract void deleteEvent(String eventID);
```

Parameters

eventID Specifies the event ID of the event to delete.

Return values

None.

Exceptions

`DeleteFailedException`
: Thrown when the underlying application's attempt to delete the event from the event store has failed.

Notes

The `deleteEvent()` method is used mainly during archiving. It deletes the event from the event store after this event has been successfully moved to the application's archive store.

Important: The `deleteEvent()` method is an abstract method.Therefore, the event-store class *must* implement this method to provide the ability to delete an event from the event store.

See also

archiveEvent()

---

fetchEvents()

Retrieves a specified number of Ready-for-Poll events from the application's event store.

Syntax

```
public abstract void fetchEvents();
```

Parameters

None.

Return values

None.

Exceptions

`StatusChangeFailedException`
: Thrown if the underlying application throws an exception while fetching the event from its event store.

Notes

The `fetchEvents()` method searches the event store for event records with the `READY_FOR_POLL` status and puts them in the even. The number of events that `fetchEvents()` retrieves is determined by the value of the `PollQuantity` connector configuration property. For each retrieved event, the method must create a `CWConnectorEvent` event object and put this event object into a Java `Vector`. The method then calls the `setEventsToProcess()` to save this event vector into the `eventsToProcess` vector, which is a member of the `CWConnectorEventStore` object. The `fetchEvents()` method determines the order in which event objects are stored in the `eventsToProcess` vector.

Important: The `fetchEvents()` method is an abstract method. Therefore, the event-store class *must* implement this method to provide the ability to fetch `READY_FOR_POLL` events from the event store.

Note: The `fetchEvents()` method is usually called from the poll method, `pollForEvents()`.

See also

getNextEvent(),, pollForEvents(),, setEventsToProcess()

---

getBO()

Builds a business object based on the information in an event from the event store.

Syntax

```
public CWConnectorBusObj getBO(CWConnectorEvent eventObject);
public CWConnectorBusObj getBO(CWConnectorEvent eventObject, int status);
```

Parameters

eventObject Is the event that contains the business object information.

status Is a status value set by some method or exception within the `getBO()` method.

Return values

A `CWConnectorBusObj` object containing a new business object based on information retrieved from the application's database. If the method was unable to retrieve the *`eventObject`* event object, it returns `null`.

Exceptions

`AttributeNotFoundException`
Thrown if `getBO()` cannot find an attribute when assigning a key value to a key attribute.

`SpecNameNotFoundException`
Thrown if the name of the business object within the event object is invalid.

`AttributeValueException`
Thrown if the retrieved attribute value is not valid for a particular attribute.

`InvalidVerbException`
Thrown if the verb within the event object is invalid.

`WrongAttributeException`
Thrown if `getBO()` encounters an invalid attribute type when assigning a key value a key attribute. For example, if the attribute is a container, it cannot hold a key value.

`AttributeNullValueException`
Thrown if the business object could not be created.

Notes

The `getBO()` method returns a business object that contains information for an application entity that the *`eventObject`* event object describes.

Important: The `getBO()` method must be overridden by *any* connector that does *not* use the RetrieveByContent verb. It must also be overridden if you want to return an internal status code to the calling method.

The default implementation of this method performs the following actions:

- Create a temporary `CWConnectorBusObj` object to hold the new business object.
- Populate the `CWConnectorBusObj` object with the data and key values from the *`eventObject`* event object.
- Take one of the following actions, based on the value of the verb in the event object:

Verb	getBO() action taken
Delete	Do *not* retrieve the object with `doVerbFor()`.
Create, Update	Set the business object's verb to RetrieveByContent and call the `doVerbFor()` method (in the `CWConnectorBusObj` class) to retrieve the remaining information from the application.

  If the verb is Create or Update, populate the `CWConnectorBusObj` object with the data that `doVerbFor()` has retrieved. It handles the following conditions that the `doVerbFor()` method might generate:
  - If `doVerbFor()` does not find the specified entity in the application, it returns `BO_DOES_NOT_EXIST`. In this case, `getBO()` sets the event status of *`eventObject`* to `ERROR_OBJECT_NOT_FOUND` and returns `null`.
  - If `doVerbFor()` is not able to connect to the application, it returns `APPRESPONSETIMEOUT`. In this case, `getBO()` calls the `setTerminate()` method (in the `CWConnectorEventStore` class) to set the internal terminate-connector flag. For more information, see "Retrieving application data" on page 187..
  - If `doVerbFor()` returns some other error (such as `RETRIEVEBYCONTENT_FAILED`), The `getBO()` method returns `null`.
- Send the `CWConnectorBusObj` object to the connector framework by calling the `gotApplEvent()` method.

Note: The `getBO()` method is usually called from the poll method, `pollForEvents()`.

As described above, the default implementation of `getBO()` has several ways to indicate to the calling method that certain error or exception conditions occur. However, if you need to return a particular internal status value (such as the status attribute of a thrown exception) to the calling method, you can override this default implementation. For your implementation of `getBO()`, use the second form of this method's signature, which provides a *status* argument. Within `getBO()`, assign some status value to this argument before you exit `getBO()`. From the calling method, pass in the uninitialized status value and, after the call to `getBO()`, access the initialized status value.

Note: The default implementation of the `pollForEvents()` method calls the first form of `getBO()`; that is, it does *not* handle any initialized status value returned by `getBO()`.

See also

doVerbFor(),, getTerminate(), pollForEvents(), setTerminate()

getNextEvent()

Retrieves the next event object from the eventsToProcess vector.

Syntax

```
public CWConnectorEvent getNextEvent();
```

Parameters

None.

Return values

A CWConnectorEvent object for the next Ready-for-Poll event. If the eventsToProcess vector is empty, the method returns null.

Exceptions

InvalidStatusChangeException
: Thrown when the event status is being changed to an invalid status value for the application.

StatusChangeFailedException
: Thrown when the status change from READY_FOR_POLL to IN_PROGRESS fails.

Notes

The getNextEvent() method checks the eventsToProcess vector for events that currently have the READY_FOR_POLL status. If it finds the such an event in this vector, the method takes the following actions:

1. Get the next event to process from the eventsToProcess vector. The fetchEvents() method determines the order in which event objects are stored in the eventsToProcess vector.
2. Change its event status to IN_PROGRESS.
3. Return the event to the caller.

The eventsToProcess vector is initialized with either the fetchEvents() or setEventsToProcess() method.

Note: The getNextEvent() method is usually called from the poll method, pollForEvents().

See also

fetchEvents(),, pollForEvents(),, setEventsToProcess()

getTerminate()

Retrieves the value of the internal terminate-connector flag.

Syntax

```
public boolean getTerminate();
```

Parameters

None.

Return values

Aboolean value that indicates the current setting of the internal terminate-connector flag.

Exceptions

None.

Notes

The `getTerminate()` method retrieves the value of an internal flag that indicates that the connector framework should terminate the connector. The connector can set the status of this internal flag with the `setTerminate()` method. The `pollForEvents()` method should call the `getTerminate()` method after its call to `getBO()` to determine whether to return the `APPRESPONSETIMEOUT` outcome status. For more information, see "Retrieving application data" on page 187..

See also

getBO(),, setTerminate()

---

recoverInProgressEvents()

Recovers any In-Progress events in the event store.

Syntax

```
public abstract int recoverInProgress();
```

Parameters

None.

Return values

An integer that indicates the outcome status of the recovery operation. Compare this integer value with the following outcome-status constants to determine the status:

`CWConnectorConstant.SUCCEED`
: The recovery of in-progress events succeeded.

`CWConnectorConstant.FAIL`
: The recovery of in-progress events failed.

Exceptions

`InvalidStatusChangeException`
: Thrown when the status is being changed to an invalid status value for the application.

`StatusChangeFailedException`
: Thrown when the status change from `IN_PROGRESS` to `READY_FOR_POLL` fails.

`AttributeNullValueException`
: Thrown if the `InDoubtEvents` connector configuration property is not defined and set.

Notes

The recoverInProgressEvents() method checks the event store for any events that currently have the IN_PROGRESS status. An event might remain in the event store with an event status of IN_PROGRESS if the connector was unexpectedly shutdown.

Note: The CWConnectorEventStore class does *not* provide a default implementation for the recoverInProgressEvents() method. Therefore, the event-store class *must* implement this method to provide the ability to recover In-Progress events at connector startup.

One possible way to implement recoverInProgressEvents() is to base its actions on the InDoubtEvents connector configuration property. If such events exist, the method can take one of the following actions, based on the value of this property:

Value of InDoubtEvents	Action for recoverInProgressEvents()
Reprocess	Change all events with the IN_PROGRESS status to the READY_FOR_POLL status so that they are sent to the connector framework in subsequent poll calls.
FailOnStartup	Log a fatal error and return a FAIL outcome status to agentInit(), which in turn throws the InProgressEventRecoveryFailedException exception. This action also sends an automatic email, if LogAtInterchangeEnd is set to True.
LogError	Log a fatal error but do *not* return FAIL outcome status to agentInit().
Ignore	Ignore the In-Progress events.

Note: For recoverInProgressEvents() to work as described, the InDoubtEvents connector configuration property *must* be defined. If InDoubtEvents is *not* defined, recoverInProgressEvents() should throw the AttributeNullValueException exception.

The recoverInProgressEvents() methods is usually called as part of the connector initialization process, from within the agentInit() method. The agentInit() should check for the status from recoverInProgressEvents() and catch any exceptions as well. The agentInit() method should throw an exception in either of the following cases:

- If recoverInProgressEvents() returns a FAIL outcome status
- If recoverInProgressEvents() catches an exception

See also

agentInit()

resubmitArchivedEvents()

Copies the specified archived event from the application's archive store to the application's event store and changes the event status to READY_FOR_POLL.

Syntax

```
public abstract int resubmitArchivedEvents(String eventID);
```

Parameters

eventID Is the event ID for the event to resubmit.

Return values

An integer that indicates the number of events archived. If nothing is resubmitted, return a zero (0).

Exceptions

`InvalidStatusChangeException`
Thrown when the status is being changed to an invalid status value for the application.

`StatusChangeFailedException`
Thrown when the status change to `READY_FOR_POLL` fails.

Notes

The `resubmitArchivedEvents()` method resubmits unprocessed events in the archive store to the event store, where they can be processed. An event is moved to the archive store when it has no subscriptions or after it has been processed. Archiving processed or unsubscribed events ensures that events are not lost. Setting the event status to `READY_FOR_POLL` ensures that the events will be picked up on subsequent polls of the event store.

Note: The `resubmitArchivedEvents()` method is an abstract method. Therefore, the event-store class *must* implement this method to provide the ability to resubmit archived events for subsequent polls of the event store.

---

setEventStatus()

Sets the status of an event in the event store.

Syntax

```
public abstract void setEventStatus(String eventID, int status);
```

Parameters

eventID Is the event ID of the event whose status is changed.

status Is one of the following event-status constants to identify the new status of the specified event:

```
CWConnectorEventStatus.READY_FOR_POLL
CWConnectorEventStatus.IN_PROGRESS
CWConnectorEventStatus.SUCCESS
CWConnectorEventStatus.UNSUBSCRIBED
CWConnectorEventStatus.ERROR_POSTING_EVENT
CWConnectorEventStatus.ERROR_OBJECT_NOT_FOUND
CWConnectorEventStatus.ERROR_PROCESSING_EVENT
```

Return values

None.

Exceptions

`InvalidStatusChangeException`
Thrown when the status is being changed to an invalid *status* value for the application.

Notes

The `setEventStatus()` method performs the following actions:

- Check if the `status` value is valid, throwing the `InvalidStatusChangeException` exception if it is not.
- Change the status of the event identified by `eventID` in the application's event store.

Important: The `setEventStatus()` method is an abstract method.Therefore, the event-store class *must* implement this method to provide the ability to set the status of an event in the event store.

The connector must ensure that the change in event status is committed in the underlying application.

See also

updateEventStatus()

setEventsToProcess()

Sets the `eventsToProcess` vector with specified events.

Syntax

```
public void setEventsToProcess(Vector eventsVector);
```

Parameters

eventsVector Is a `Java.util.Vector` object that contains the events to process.

Return values

None.

Exceptions

None.

Notes

The `setEventsToProcess()` method assigns to the `eventsToProcess` vector of the `CWConnectorEventStore` object the contents of the *eventsVector* vector.

setTerminate()

Sets the internal terminate-connector flag to true.

Syntax

```
public void setTerminate();
```

Parameters

None.

Return values

None.

Exceptions

None.

Notes

The `setTerminate()` method sets an internal flag that tells the connector framework to terminate the connector. The connector can check the status of this internal flag with the `getTerminate()` method. The `getBO()` method should call the `setTerminate()` method after its call to `doVerbFor()` if `doVerbFor()` has returned the `APPRESPONSETIMEOUT` outcome status. For more information, see "Retrieving application data" on page 187.

See also

getTerminate()

---

updateEventStatus()

Updates the event status both in the application's event store and in the event.

Syntax

```
public void updateEventStore(CWConnectorEvent eventObject,
      int status);
```

Parameters

eventObject Is the event object whose status is updated.

status Is one of the following event-status constants to store in the event object:

```
CWConnectorEventStatus.READY_FOR_POLL
CWConnectorEventStatus.IN_PROGRESS
CWConnectorEventStatus.SUCCESS
CWConnectorEventStatus.UNSUBSCRIBED
CWConnectorEventStatus.ERROR_POSTING_EVENT
CWConnectorEventStatus.ERROR_OBJECT_NOT_FOUND
CWConnectorEventStatus.ERROR_PROCESSING_EVENT
```

Return values

None.

Exceptions

`InvalidStatusChangeException`
Thrown when the status is being changed to an invalid status value for the application.

`StatusChangeFailedException`
Thrown when the underlying application is unable to change the event status in the event store.

Notes

The `updateEventStatus()` method sets the status of the *eventObject* event to *status*. It also updates the event status within the *eventObject* event to *status*.

Deprecated Methods

Some methods in the `CWConnectorEventStore` class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but IBM recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 131 lists the deprecated methods for the `CWConnectorEventStore` class. If you are writing a new connector (not modifying an existing connector), you can ignore this section.

Table 131. Deprecated methods of the CWConnectorEventStore class

Deprecated method	Replacement
`setEventStoreStatus()`	setEventStatus()

Chapter 18. CWConnectorEventStoreFactory interface

The CWConnectorEventStoreFactory interface defines the functionality for the event-store factory, which creates an event store. If your Java connector uses an extension of the CWConnectorEventStore class to access the event store, you must create an event-store-factory class to implement the CWConnectorEventStoreFactory interface. This interface contains a method to instantiate an event-store (CWConnectorEventStore) object.

Important: All Java connectors that use an extension of the CWConnectorEventStore class to access the event store *must* provide an implementation of this interface. In this event-store-factory class, you must implement the getEventStore() method to be able to access the event store through the CWConnectorEventStore class.

Table 132 summarizes the methods in the CWConnectorEventStoreFactory interface.

Table 132. Member method of the CWConnectorEventStoreFactory interface

Member method	Description	Page
getEventStore()	Creates a new event-store object.	333

getEventStore()

Creates a new event-store object.

Syntax

```
public Object getEventStore();
```

Parameters

None.

Return values

An Object containing the newly created event-store object. If the event store cannot be located, the method returns null.

Exceptions

None.

Notes

The getEventStore() method is the event-store factory. It needs to build the corresponding event store for the connector and return the event-store object. Connectors that use more than one event store must provide implementations for this method for *each* event-store class.

The default implementation of the getEventStore() method in the CWConnectorAgent class calls the getEventStore() method of the event-store-factory class named in the EventStoreFactory connector configuration property. For more information, see "CWConnectorEventStoreFactory interface" on page 178..

See also

getEventStore()

Chapter 19. CWConnectorExceptionObject class

The `CWConnectorExceptionObject` class represents an *exception-detail object*, which provides detailed information about an exception. Each exception that methods of the Java connector library can throw can contain an exception-detail object. This class provides methods to store and access information about the exception message. Table 133 summarizes the methods in the `CWConnectorExceptionObject` class.

Table 133. Member methods of the CWConnectorExceptionObject class

Member method	Description	Page
CWConnectorExceptionObject()	Creates an exception-detail object.	335
getExpl()	Retrieves the explanation for the message associated with the exception-detail object's message number.	335
getMsg()	Retrieves the message text from an exception-detail object.	336
getMsgNumber()	Retrieves the message number (ID) associated with the message in the exception-detail object.	336
getMsgType()	Retrieves the message type associated with the message in the exception-detail object.	337
setExpl()	Sets the explanation for the message in the exception-detail object.	338
setMsg()	Sets the message text for the exception-detail object.	338
setMsgNumber()	Sets the message number (ID) associated with the message in the exception-detail object.	339
setMsgType()	Sets the message type associated with the message in the exception-detail object.	339
setStatus()	Sets the status value for the exception-detail object.	340

CWConnectorExceptionObject()

Creates an exception-detail object.

Syntax

```
public CWConnectorExceptionObject();
```

Parameters

None.

Return values

A `CWConnectorExceptionObject` object containing the newly created exception-detail object.

getExpl()

Retrieves the explanation for the message associated with the exception-detail object's message number.

Syntax

```
public String getExpl();
```

Parameters

None.

Return values

A `String` object containing the message explanation from the current exception-detail object.

Exceptions

None.

See also

setExpl()

getMsg()

Retrieves the message text from an exception-detail object.

Syntax

```
public String getMsg();
```

Parameters

None.

Return values

A `String` object that contains the message text from the current exception-detail object.

Exceptions

None.

See also

setMsg()

getMsgNumber()

Retrieves the message number (ID) associated with the message in the exception-detail object.

Syntax

```
public int getMsgNumber();
```

Parameters

None.

Return values

The integer message number of the exception-detail object's message.

See also

setMsgNumber()

getMsgType()

Retrieves the message type associated with the message in the exception-detail object.

Syntax

```
public int getMsgType();
```

Parameters

None.

Return values

The integer that indicates the message type of the exception-detail object's message. Compare this integer value with the following message-type constants to determine the message type:

```
    XRD_ERROR
XRD_FATAL
```

These message-type constants are defined in both the `CWConnectorUtil` and `CWConnectorLogAndTrace` classes.

See also

setMsgType()

getStatus()

Retrieves the status from the exception-detail object.

Syntax

```
public int getStatus();
```

Parameters

None.

Return values

An integer value that represents the status exception-detail object. Compare this integer value with the following outcome-status constants to determine the message type:

```
    CWConnectorConstant.APPRESPONSETIMEOUT
CWConnectorConstant.BO_DOES_NOT_EXIST
CWConnectorConstant.    MULTIPLE_HITS
CWConnectorConstant.RETRIEVEBYCONTENT_FAILED
    CWConnectorConstant.UNABLETOLOGIN
```

These outcome-status constants are defined in the `CWConnectorConstant` class.

Exceptions

None.

See also

setStatus()

setExpl()

Sets the explanation for the message in the exception-detail object.

Syntax

```
public void setExpl(String msgExpl);
```

Parameters

msgExpl Is a `String` object that contains the message explanation to assign to the exception-detail object.

Return values

None.

Exceptions

None.

See also

getExpl()

setMsg()

Sets the message text for the exception-detail object.

Syntax

```
public void setMsg(String newMsg);
```

Parameters

newMsg Is a `String` object that contains the message text to assign to the exception-detail object.

Return values

None.

Exceptions

None.

See also

getMsg()

setMsgNumber()

Sets the message number (ID) associated with the message in the exception-detail object.

Syntax

```
public void setMessageNumber(int msgNumber);
```

Parameters

msgNumber Is the integer message number to set for the exception-detail object's message.

Return values

None.

Exceptions

None.

See also

getMsgNumber()

setMsgType()

Sets the message type associated with the message in the exception-detail object.

Syntax

```
public void setMsgType(int msgType);
```

Parameters

msgType Is the message type that indicates the severity of the message in the exception-detail object. Use one of the following message-type constants:

```
XRD_ERROR
XRD_FATAL
```

Note: Even though other message-type constants exist, they are *not* valid as types for a message in the exception-detail object. This object is part of the exception object, which is only thrown when an exception occurs.

Return values

None.

Exceptions

None.

See also

getMsgType()

setStatus()

Sets the status value for the exception-detail object.

Syntax

```
public void setStatus(int status);
```

Parameters

status Is an integer value that indicates the outcome status to assign to the exception-detail object.

Return values

None.

Notes

You must set the exception status of an exception-detail object with the `setStatus()` method *before* the exception is thrown. This status value allows the calling code to take appropriate action to cleanup any application-related resources (for example from an `APPRESPONSETIMEOUT` status) before to passing this status back to the connector framework.

Exceptions

None.

See also

getStatus()

Chapter 20. CWConnectorLogAndTrace class

The CWConnectorLogAndTrace class defines the log-trace constants shared by all connectors. This class contains the following static constants:

- "Message-type constants"
- "Trace-level constants"

Message-type constants

Table 134 summarizes the static message-type constants, which are defined in the CWConnectorLogAndTrace cclass.

Table 134. Message-type constants of the CWConnectorLogAndTrace class

Constant name	Meaning
XRD_WARNING	A warning message
XRD_TRACE	A trace message
XRD_INFO	An informational message
XRD_ERROR	An error message
XRD_FATAL	A fatal error message

Trace-level constants

Table 135 summarizes the static trace-level constants, which are defined in the CWConnectorLogAndTrace cclass.

Table 135. Trace-level constants of the CWConnectorLogAndTrace class

Constant name	Meaning
LEVEL0	Level 0 of tracing (turn tracing off)
LEVEL1	Level 1 of tracing
LEVEL2	Level 2 of tracing
LEVEL3	Level 3 of tracing
LEVEL4	Level 4 of tracing
LEVEL5	Level 5 of tracing

Chapter 21. CWConnectorReturnStatusDescriptor class

The CWConnectorReturnStatusDescriptor class enables Java connectors to return error and informational messages in a return-status descriptor. This descriptor provides additional status information is usually returned as part of the request response sent to the integration broker that initiated the request.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework returns the return-status descriptor to the collaboration that initiated the request. The collaboration can access the information in this return-status descriptor to obtain the status of its service call request.

Note: The CWConnectorReturnStatusDescriptor class extends the ReturnStatusDescriptor class of the low-level Java connector library. For more information on the classes of the low-level Java connector library, see Chapter 26, "Overview of the low-level Java connector library," on page 405.

Table 136 summarizes the methods in the CWConnectorReturnStatusDescriptor class.

Table 136. Member methods of the CWConnectorReturnStatusDescriptor class

Member method	Description	Page
CWConnectorReturnStatusDescriptor()	Creates a return-status descriptor.	343
getErrorString()	Retrieves a message string from a return-status descriptor.	344
getStatus()	Retrieves the value of the status code from the return-status descriptor.	344
setErrorString()	Sets the error or informational message in the return-status descriptor.	344
setStatus()	Sets the value of the status code in the return-status descriptor.	345

CWConnectorReturnStatusDescriptor()

Creates a return-status descriptor.

Syntax

```
public CWConnectorReturnStatusDescriptor();
```

Parameters

None.

Return values

A CWConnectorReturnStatusDescriptor object containing the newly created return-status descriptor.

getErrorString()

Retrieves a message string from a return-status descriptor.

Syntax

```
public String getErrorString();
```

Parameters

None.

Return values

A `String` containing an error or informational message for the integration broker, or `null`.

Exceptions

None.

Notes

The `getErrorString()` method returns a message that can be an error message or an informational message.

See also

setErrorString()

getStatus()

Retrieves the value of the status code from the return-status descriptor.

Syntax

```
public int getStatus();
```

Parameters

None.

Return values

An `int` value indicating the status of an operation.

Exceptions

None.

See also

setStatus()

setErrorString()

Sets the error or informational message in the return-status descriptor.

Syntax

```
public void setErrorString(String errorStr);
```

Parameters

errorStr Is the value to set the message string.

Return values

None.

Exceptions

None.

See also

getErrorString()

setStatus()

Sets the value of the status code in the return-status descriptor.

Syntax

```
public void setStatus(int status);
```

Parameters

status Is the value of status code to assign to the return-status descriptor.

Return values

None.

Exceptions

None.

See also

getStatus()

Chapter 22. CWConnectorUtil class

The CWConnectorUtil class contains miscellaneous utility methods.

Note: The CWConnectorUtil class extends the JavaConnectorUtil class of the low-level Java connector library. For more information on the classes of the low-level Java connector library, see Chapter 26, "Overview of the low-level Java connector library," on page 405.

This class contains the following:

- "Message-file constants"
- "Methods"

Message-file constants

Table 137 summarizes the static message-file constants, which are defined in the CWConnectorUtil class.

Table 137. Message-file constants of the CWConnectorUtil class

Constant name	Meaning
CONNECTOR_MESSAGE_FILE	Use the connector message file to generate messages.
INFRASTRUCTURE_MESSAGE_FILE	Use the InterChange Server message file (InterchangeSystem.txt) to generate messages. **Important:** Connectors should *not* obtain messages from the InterchangeSystem.txt file. Instead, they should always use their local connector message file.

Methods

The CWConnectorUtil class contains miscellaneous utility methods for use in a Java connector. These utility methods fall into the following general categories:

- Static methods for generating and logging messages
- Static methods for creating business objects
- Static methods for obtaining connector configuration properties
- Methods for obtaining locale information

Table 138 summarizes the methods in the CWConnectorUtil class.

Table 138. Member methods of the CWConnectorUtil class

Member method	Description	Page
CWConnectorUtil()	Creates a CWConnectorUtil object.	349
boToByteArray()	Calls a data handler to convert a business object to serialized data of a specified MIME type. This serialized data can be accessed through a byte array.	349

Table 138. Member methods of the CWConnectorUtil class (continued)

Member method	Description	Page
boToStream()	Calls a data handler to convert a business object to serialized data of a specified MIME type. This serialized data can be accessed through an input stream.	351
boToString()	Calls a data handler to convert a business object to serialized data of a specified MIME type. This serialized data can be accessed as a string.	353
byteArrayToBo()	Calls a data handler to convert serialized data of a specified MIME type to a business object. This serialized data is accessed through a byte array.	355
createAndCopyKeyVals()	Creates a new business object, assigning it the specified key values and verb and default values to the remaining attributes.	356
createAndSetDefaults()	Creates a new business object, assigning default values to all its attributes.	357
createBusObj()	Creates a new business object.	358
generateAndLogMsg()	Generates a message and sends it to the log destination.	358
generateAndTraceMsg()	Generates a trace message and sends it to the log destination.	360
generateMsg()	Generates a message from a set of predefined messages in a message file.	361
getAllConfigProperties()	Retrieves a list of all connector configuration properties, regardless of whether the property is simple, hierarchical, or multi-valued.	362
getAllConnectorAgentProperties()	Retrieves a list of all connector configuration properties for the current connector. However, it retrieves them as single-valued properties.	363
getBlankValue()	Retrieves the value for the special Blank attribute value.	364
getConfigProp()	Retrieves the value of a connector configuration property.	364
getCWConnectorAPIVersion()	Retrieves the version of the Java connector library.	364
getGlobalEncoding()	Retrieves the character encoding that the connector framework is using.	365
getGlobalLocale()	Retrieves the locale of the connector framework.	366
getHierarchicalConfigProp()	Retrieves the value of a hierarchical connector configuration property.	366
getIgnoreValue()	Retrieves the value for the special “Ignore” attribute value.	367
getSupportedBONames()	Retrieves a list of supported business objects for the current connector.	368
getVersion()	Retrieves the version of the connector.	368
initAndValidateAttributes()	Initializes attributes by setting them to their default values and for each attribute and then validates the attributes.	369
isBlankValue()	Determines if an attribute value is the special Blank value.	370
isIgnoreValue()	Determines if an attribute value is the special Ignore value.	370
isTraceEnabled()	Determines if the trace level is greater than or equal to the trace level for which it is looking, if tracing is enabled at this level.	371
logMsg()	Logs a message to the connector’s log destination.	371

Table 138. Member methods of the CWConnectorUtil class (continued)

Member method	Description	Page
readerToBO()	Calls a data handler to convert serialized data of a specified MIME type to a business object. This serialized data is accessed with a Reader object.	372
streamToBO()	Calls a data handler to convert serialized data of a specified MIME type to a business object. This serialized data is accessed through an input stream.	374
stringToBo()	Calls a data handler to convert serialized data of a specified MIME type to a business object. This serialized data is accessed as a string.	376
traceCWConnectorAPIVersion()	Traces the Java connector library version at a trace level 1.	378
traceWrite()	Writes a trace message to the log destination.	378

CWConnectorUtil()

Creates a `CWConnectorUtil` object.

Syntax

```
public CWConnectorUtil();
```

Parameters

None.

Return values

A `CWConnectorUtil` object.

boToByteArray()

Calls a data handler to convert a business object to serialized data of a specified MIME type. This serialized data can be accessed as a byte array.

Syntax

```
public static byte[] boToByteArray(CwConnectorBusObj theBusObj, String mimeType,
      String BOPrefix, String encoding, Locale locale, Object config);
```

Parameters

BOPrefix Is the optional business-object prefix, which is combined with *mimeType* to form the key of the child meta-object. This argument can be used to specify a MIME subtype. It can also be used to specify a value for the `BOPrefix` data-handler configuration property.

config Is an `Object` that contains additional configuration information for the data handler.

encoding Specifies the character encoding for the serialized data in the `byte` array. If you specify `null`, the method uses the character encoding of the machine.

locale Is a `java.util.Locale` object that specifies the locale for the serialized data in the `byte` array. If you specify `null`, the method uses the connector-framework locale.

mimeType Is the MIME type that identifies the serialized format to which to convert the business object.

theBusObj Is the business object to serialize to the specified MIME type and return a `byte` array.

Return values

A `byte` array that contains the serialized business object in the specified MIME type.

Exceptions

`DataHandlerCreateException`
Thrown when the `boToByteArray()` method cannot instantiate a data handler for the specified MIME type.

`ParseException`
Thrown when the data handler encounters some error during the conversion of the business object to the specified MIME type.

`PropertyNotSetException`
Thrown when the `DataHandlerMetaObjectName` connector configuration property is not set.

The `boToByteArray()` method can also throw the general Java exception `NullPointerException` if the data handler returns a `null` pointer instead of a `byte` array.

Notes

The `boToByteArray()` method provides the connector with the ability to call a data handler to perform business-object-to-string conversion. With this method, the resulting serialized data can be accessed through a Java `byte` array. The method identifies which data handler to invoke based on the specified *mimeType* argument. It instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object, as follows:

- It checks the top-level meta-object for the data handler that corresponds to this MIME type. It obtains the name of this top-level meta-object from the `DataHandlerMetaObjectName` connector configuration property. If this property is not set, `boToByteArray()` throws the `PropertyNotSetException` exception.
- The instantiation process converts the specified *mimeType* to its equivalent MIME-type string and then searches the top-level meta-object for an attribute whose name matches this MIME-type string. The associated type for this attribute is the child meta-object.
- It obtains the name of the class to instantiate from the `ClassName` attribute in the child meta-object.

  If `boToByteArray()` specifies a *BOPrefix* and a *mimeType*, it instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object. However, when a *BOPrefix* is specified, the instantiation process interprets this value as a MIME subtype. It searches the top-level meta-object for an attribute whose name includes *both* the MIME type and subtype.

If the data handler cannot be instantiated, `boToByteArray()` throws the `DataHandlerCreateException`. For more information on how the arguments of `boToByteArray()` identify which data handler to instantiate, see "Identifying the data handler to instantiate" on page 77.

Once instantiated, the data handler converts the specified business object to the serialized format that the MIME type indicates. If `boToByteArray()` specifies the *encoding* and *locale* arguments, the data handler uses the specified character encoding and locale when it creates the serialized data. The data handler returns this serialized data to the `boToByteArray()` method as a `byte` array, through which the calling method can access the returned serialized data.

Note: If the data handler cannot convert the business object, `boToByteArray()` throws the `ParseException` exception.

You can specify a *config* option if you need to provide the data handler with more configuration information than is available in its meta-object. This argument can be used to specify a template file or a string to a URL for a schema that is used to build an XML document from a business object.

See also

boToStream(), boToString(), byteArrayToBo()

boToStream()

Calls a data handler to convert a business object to serialized data of a specified MIME type. This serialized data can be accessed through an input stream.

Syntax

```
public static InputStream boToStream(CWConnectorBusObj theBusObj, String mimeType);
public static InputStream boToStream(CwConnectorBusObj theBusObj, String mimeType,
      Object config);
public static InputStream boToStream(CwConnectorBusObj theBusObj, String mimeType,
      String BOPrefix, String encoding, Locale locale, Object config);
```

Parameters

BOPrefix Is the optional business-object prefix, which is combined with *mimeType* to form the key of the child meta-object. This argument can be used to specify a MIME subtype. It can also be used to specify a value for the `BOPrefix` data-handler configuration property.

config Is an `Object` that contains additional configuration information for the data handler.

encoding Specifies the character encoding for the serialized data in the input stream. If you specify `null`, the method uses the character encoding of the machine.

locale Is a `java.util.Locale` object that specifies the locale for the serialized data in the input stream. If you specify `null`, the method uses the connector-framework locale.

mimeType Is the MIME type that identifies the serialized format to which to convert the business object.

theBusObj Is the business object to serialize tothe specified MIME type and return an input stream.

Return values

An object of the Java `java.io.InputStream` class (or one of its subclasses) that contains the serialized business object in the specified MIME type.

Exceptions

`DataHandlerCreateException`
: Thrown when the `boToStream()` method cannot instantiate a data handler for the specified MIME type.

`ParseException`
: Thrown when the data handler encounters some error during the conversion of the business object to the specified MIME type.

`PropertyNotSetException`
: Thrown when the `DataHandlerMetaObjectName` connector configuration property is not set.

The `boToStream()` method can also throw the general Java exception `NullPointerException` if the data handler returns a `null` pointer instead of an `InputStream` object.

Notes

The `boToStream()` method provides the connector with the ability to call a data handler to perform business-object-to-string conversion. With this method, the resulting serialized data can be accessed through a Java input stream (based on the `InputStream` class). The method identifies which data handler to invoke based on the specified *mimeType* argument. It instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object, as follows:

- It checks the top-level meta-object for the data handler that corresponds to this MIME type. It obtains the name of this top-level meta-object from the `DataHandlerMetaObjectName` connector configuration property. If this property is not set, `boToStream()` throws the `PropertyNotSetException` exception.
- The instantiation process converts the specified *mimeType* to its equivalent MIME-type string and then searches the top-level meta-object for an attribute whose name matches this MIME-type string. The associated type for this attribute is the child meta-object.
- It obtains the name of the class to instantiate from the `ClassName` attribute in the child meta-object.

  If `boToStream()` specifies a *BOPrefix* and a *mimeType*, it instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object. However, when a *BOPrefix* is specified, the instantiation process interprets this value as a MIME subtype. It searches the top-level meta-object for an attribute whose name includes *both* the MIME type and subtype.

If the data handler cannot be instantiated, `boToStream()` throws the `DataHandlerCreateException`. For more information on how the arguments of `boToStream()` identify which data handler to instantiate, see "Identifying the data handler to instantiate" on page 77.

Once instantiated, the data handler converts the specified business object to the serialized format that the MIME type indicates. If `boToStream()` specifies the *encoding* and *locale* arguments, the data handler uses the specified character encoding and locale when it creates the serialized data. The data handler returns this serialized data to the `boToStream()` method as an input stream, through which the calling method can access the returned serialized data.

Note: If the data handler cannot convert the business object, `boToStream()` throws the `ParseException` exception.

You can specify a *config* option if you need to provide the data handler with more configuration information than is available in its meta-object. This argument can be used to specify a template file or a string to a URL for a schema that is used to build an XML document from a business object.

See also

boToByteArray(), boToString(), streamToBO()

boToString()

Calls a data handler to convert a business object to serialized data of a specified MIME type. This serialized data can be accessed as a string.

Syntax

```
public static String boToString(CWConnectorBusObj theBusObj, String mimeType);
public static String boToString(CwConnectorBusObj theBusObj, String mimeType,
      Object config);
public static String boToString(CwConnectorBusObj theBusObj, String mimeType,
      String BOPrefix, String encoding, Object config);
```

Parameters

BOPrefix Is the business-object prefix, which is combined with *mimeType* to form the key of the child meta-object. This argument can be used to specify a MIME subtype. It can also be used to specify a value for the `BOPrefix` data-handler configuration property.

config Is an `Object` that contains additional configuration information for the data handler.

encoding Specifies the character encoding for the serialized data in the `String`. If you specify `null`, the method uses the character encoding of the machine.

locale Is a `java.util.Locale` object that specifies the locale for the serialized data in the `String`. If you specify `null`, the method uses the connector-framework locale.

mimeType Is the MIME type that identifies the serialized format to which to convert the business object.

theBusObj Is the business object to serialize to the specified MIME type and return a string.

Return values

A `String` object that contains the serialized business object in the specified MIME type.

Exceptions

`DataHandlerCreateException`
Thrown when the `boToString()` method cannot instantiate a data handler for the specified MIME type.

`ParseException`
Thrown when the data handler encounters some error during the conversion of the business object to the specified MIME type.

`PropertyNotSetException`
Thrown when the `DataHandlerMetaObjectName` connector configuration property is not set.

The boToString() method can also throw the general Java exception NullPointerException if the data handler returns a null pointer instead of a String object.

Notes

The boToString() method provides the connector with the ability to call a data handler to perform business-object-to-string conversion. With this method, the resulting serialized data can be accessed through a Java String. The method identifies which data handler to invoke based on the specified *mimeType* argument. It instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object, as follows:

- It checks the top-level meta-object for the data handler that corresponds to this MIME type. It obtains the name of this top-level meta-object from the DataHandlerMetaObjectName connector configuration property. If this property is not set, boToString() throws the PropertyNotSetException exception.
- The instantiation process converts the specified *mimeType* to its equivalent MIME-type string and then searches the top-level meta-object for an attribute whose name matches this MIME-type string. The associated type for this attribute is the child meta-object.
- It obtains the name of the class to instantiate from the ClassName attribute in the child meta-object.

  If boToString() specifies a *BOPrefix* and a *mimeType*, it instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object. However, when a *BOPrefix* is specified, the instantiation process interprets this value as a MIME subtype. It searches the top-level meta-object for an attribute whose name includes *both* the MIME type and subtype.

If the data handler cannot be instantiated, boToString() throws the DataHandlerCreateException. For more information on how the arguments of boToString() identify which data handler to instantiate, see "Identifying the data handler to instantiate" on page 77.

Once instantiated, the data handler converts the specified business object to the serialized format that the MIME type indicates. If boToString() specifies the *encoding* and *locale* arguments, the data handler uses the specified character encoding and locale when it creates the serialized data. The data handler returns this serialized data to the boToString() method as a String object, through which the calling method can access the returned serialized data.

Note: If the data handler cannot convert the business object, boToString() throws the ParseException exception.

You can specify a *config* option if you need to provide the data handler with more configuration information than is available in its meta-object. This argument can be used to specify a template file or a string to a URL for a schema that is used to build an XML document from a business object.

See also

boToByteArray(), boToStream(), stringToBo()

byteArrayToBo()

Calls a data handler to convert serialized data of a specified MIME type to a business object. This serialized data is accessed as a `byte` array.

Syntax

```
public static CWConnectorBusObj byteArrayToBo(CWConnectorBusObj theBusObj,
      byte[] serializedData, String mimeType, String BOPrefix,
      String encoding, Locale locale, Object config);
```

Parameters

BOPrefix Is the business-object prefix, which is combined with *mimeType* to form the key of the child meta-object. This argument can be used to specify a MIME subtype. It can also be used to specify a value for the `BOPrefix` data-handler configuration property.

config Is an `Object` that contains additional configuration information for the data handler.

encoding Specifies the character encoding for the serialized data in the `byte` array. If you specify `null`, the method uses the character encoding of the machine.

locale Is a `java.util.Locale` object that specifies the locale for the serialized data in the `byte` array. If you specify `null`, the method uses the connector-framework locale.

mimeType Is the MIME type that identifies the format of the serialized data.

serializedData Is a `byte` array that contains the serialized data to convert to a business object.

theBusObj Identifies the type of business object (business object definition) to which the method converts the serialized data.

Return values

A `CWConnectorBusObj` object that contains the business object for the serialized data.

Exceptions

`DataHandlerCreateException`
Thrown when the `byteArrayToBo()` method cannot instantiate a data handler for the specified MIME type.

`ParseException`
Thrown when the data handler encounters some error during the conversion of the serialized data to the specified business object.

`PropertyNotSetException`
Thrown when the `DataHandlerMetaObjectName` connector configuration property is not set.

The `byteArrayToBo()` method can also throw the general Java exception `NullPointerException` if the data handler returns a `null` pointer instead of a business object.

Notes

The `byteArrayToBo()` method provides the connector with the ability to call a data handler to perform string-to-business-object conversion. With this method, the incoming *serializedData* is accessed through a Java `byte` array. The method identifies which data handler to invoke based on the specified *mimeType* argument. It

instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object, as follows:

- It checks the top-level meta-object for the data handler that corresponds to this MIME type. It obtains the name of this top-level meta-object from the `DataHandlerMetaObjectName` connector configuration property. If this property is not set, `byteArrayToBo()` throws the `PropertyNotSetException` exception.
- The instantiation process converts the specified *mimeType* to its equivalent MIME-type string and then searches the top-level meta-object for an attribute whose name matches this MIME-type string. The associated type for this attribute is the child meta-object.
- It obtains the name of the class to instantiate from the `ClassName` attribute in the child meta-object.

  If `byteArrayToBo()` specifies a *BOPrefix* and a *mimeType*, it instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object. However, when a *BOPrefix* is specified, the instantiation process interprets this value as a MIME subtype. It searches the top-level meta-object for an attribute whose name includes *both* the MIME type and subtype.

If the data handler cannot be instantiated, `byteArrayToBo()` throws the `DataHandlerCreateException`. For more information on how the arguments of `byteArrayToBo()` identify which data handler to instantiate, see "Identifying the data handler to instantiate" on page 77.

Once instantiated, the data handler converts the specified serialized data to a business object of the type that *theBusObj* indicates. If `byteArrayToBo()` specifies the *encoding* and *locale* arguments, the data handler uses the specified character encoding and locale when it interprets the serialized data. The data handler returns this business object to the `byteArrayToBo()` method, which in turn returns it to the calling method.

Note: If the data handler cannot convert the serialized data, `byteArrayToBo()` throws the `ParseException` exception.

You can specify a *config* option if you need to provide the data handler with more configuration information than is available in its meta-object. This argument can be used to specify a template file or a string to a URL for a schema that is used to build an XML document from a business object.

See also

boToByteArray(), readerToBO(), streamToBO(), stringToBo()

createAndCopyKeyVals()

Creates a new business object, assigning it the specified key values and verb and default values to the remaining attributes.

Syntax

```
public static CWConnectorBusObj createAndCopyKeyVals(String busObjName,
        String keyVals, String verb, String delimiter)
```

Parameters

busObjName Is the name of the business object to create.

keyVals Is the key-value string, which is an ordered list of primary-key values separated by the *delimiter*.

verb Is the verb to assign to the new business object.

delimiter Is the key delimiter.

Return values

A `CWConnectorBusObj` object containing the newly created business object.

Exceptions

`SpecNameNotFoundException`
Thrown when the business object definition is not found for the name specified.

`AttributeNotFoundException`
Thrown when an attribute cannot be found.

`AttributeValueException`
Thrown when an attribute is set to an invalid value.

`WrongAttributeException`
Thrown when the attribute's value does not match its data type.

`InvalidVerbException`
Thrown when the verb value is invalid.

Notes

The `createAndCopyKeyVals()` method performs the following tasks:

- Create a new business object of the type specified by *busObjName*.
- Parses the *keyVals* key string to obtain the key values and sets these in the business object's key attributes. The method assumes that the key values are delimited with the specified *delimiter* value.
- Set the new business object's verb to *verb*.
- Assign default attribute values to the remaining attributes in the business object.

This method is useful in the `pollForEvents()` method to build the business objedt that is to be sent to the integration broker for further processing.

createAndSetDefaults()

Creates a new business object, assigning default values to all its attributes.

Syntax

```
public static CWConnectorBusObj createAndSetDefaults(
   String busObjName)
```

Parameters

busObjName Is the name of the business object to create.

Return values

A `CWConnectorBusObj` object containing the newly created business object.

Exceptions

`SpecNameNotFoundException`
Thrown when the business object definition is not found for the name specified.

`AttributeNotFoundException`
: Thrown when one of the business object's attributes (as defined by the business object definition) cannot be found.

Notes

The `createAndSetDefaults()` method performs the following tasks:

- Create a new business object of the type specified by *busObjName*.
- Assign default attribute values to the all attributes in the business object.

createBusObj()

Creates a new business object.

Syntax

```
public static final CWConnectorBusObj createBusObj(String busObjName);
public static final CWConnectorBusObj createBusObj(String busObjName,
    Locale localeObject);
public static final CWConnectorBusObj createBusObj(String busObjName,
    String localeName);
```

Parameters

busObjName Specifies the name of the business object to create.

localeObject Is the Java `Locale` object that identifies the locale to associate with the business object.

localeName Is the name of the locale to associate with the business object.

Return values

A `CWConnectorBusObj` object containing the newly created business object.

Exceptions

`SpecNameNotFoundException`
: Thrown when the business object definition is not found for the name specified.

Notes

The `createBusObj()` method creates a new business object instance whose type is the business object definition you specify in *busObjName*. If you specify a *localeObject* or *localeName*, this business-object locale applies to the data in the business object, *not* to the name of the business object definition or its attributes (which must be characters in the code set associated with the U.S. English locale, `en_US`). For a description of the format for *localeName*, see "Design Considerations for an Internationalized Connector," on page 54.

See also

getLocale()

generateAndLogMsg()

Generates a message and sends it to the log destination.

Syntax

```
public static void generateAndLogMsg(int msgNum, int msgType, int isGlobal);

public static void generateAndLogMsg(int msgNum, int msgType, int isGlobal,
      msgParameters);
```

Parameters

isGlobal Is the CONNECTOR_MESSAGE_FILE message-file constant defined in the CWConnectorUtil class to indicate that the message file is the connector message file.

msgNum Specifies the message number (identifier) in the message file.

msgParameters Is an optional list of String parameter values, each corresponding to a parameter in the message list, for a maximum of ten parameters.

msgType Is one of the following message-type constants defined in the CWConnectorLogAndTrace class to identify the message severity:

```
CWConnectorLogAndTrace.XRD_WARNING
CWConnectorLogAndTrace.XRD_ERROR
CWConnectorLogAndTrace.XRD_FATAL
CWConnectorLogAndTrace.XRD_INFO
CWConnectorLogAndTrace.XRD_TRACE
```

Return values

None.

Exceptions

None.

Notes

The generateAndLogMsg() method combines the message generating and logging functionality of generateMsg() and logMsg(), respectively. It generates a message from a message file and then sends it to the log destination. You establish the name of a connector's log destination through the Logging section in the Trace/Log File tab of Connector Configurator.

The method can take in variable number of string arguments. It supports up to a total of ten parameter values. That is, you can provide up to ten String values as arguments to generateAndLogMsg(). The following call provides values for seven parameters in error message 3223, which is defined in the connector message file:

```
generateAndLogMsg(3223, CWConnectorLogAndTrace:XRD_ERROR,
     CWConnectorUtil.CONNECTOR_MESSAGE_FILE,
     value1, value2, value3, value4, value5, value6, value7);
```

WebSphere InterChange Server

If severity is XRD_ERROR or XRD_FATAL and the connector configuration property LogAtInterchangeEnd is set, the error message is logged and an email notification is sent when email notification is on. See the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set for information on how to set up email notification for errors.

IBM recommends that log messages be contained in a message file and extracted with the generateAndLogMsg() method. This message file should be the connector message file, which contains messages specific to your connector.

Connector messages logged with generateAndLogMsg() are viewable using LogViewer.

See also

generateAndTraceMsg(),, generateMsg(),, logMsg()

generateAndTraceMsg()

Generates a trace message and sends it to the trace destination.

Syntax

```
public static void generateAndTraceMsg(int traceLevel, int msgNum,
    int isGlobal);

public static void generateAndTraceMsg(int traceLevel, int msgNum,
    int isGlobal, msgParameters);
```

Parameters

isGlobal Is the CONNECTOR_MESSAGE_FILE message-file constant defined in the CWConnectorUtil class to indicate that the message file is the connector message file.

msgNum Specifies the message number (identifier) in the message file.

msgParameters Is an optional list of String parameter values, each corresponding to a parameter in the message list, with a maximum of ten parameters.

traceLevel Is one of the following trace-level constants defined in the CWConnectorLogAndTrace class to identify the trace level used to determine which trace messages are output:

```
CWConnectorLogAndTrace.LEVEL1
CWConnectorLogAndTrace.LEVEL2
CWConnectorLogAndTrace.LEVEL3
CWConnectorLogAndTrace.LEVEL4
CWConnectorLogAndTrace.LEVEL5
```

The method writes the trace message when the current trace level is greater than or equal to *traceLevel*.

Note: Do not specify a trace level of zero (LEVEL0) with a tracing message. A trace level of zero indicates that tracing is turned off. Therefore, any trace message associated with a *traceLevel* of LEVEL0 will never print.

Return values

None.

Exceptions

None.

Notes

The generateAndTraceMsg() method combines the message generating and tracing functionality of generateMsg() and traceWrite(), respectively. It generates a message from a message file and then sends it to the trace destination. You establish the name of a connector's trace destination through the Tracing section in the Trace/Log File tab of Connector Configurator.

The method can take in variable number of string arguments. It supports up to a total of ten parameter values. That is, you can provide up to ten `String` values as arguments in the *msgParameters* parameter of `generateAndTraceMsg()`. The following call provides values for seven parameters in trace message 668, which is defined in the connector message file:

```
generateAndTraceMsg(CWConnectorLogAndTrace.LEVEL3, 668,
        CWConnectorUtil.CONNECTOR_MESSAGE_FILE,
        value1, value2, value3, value4, value5, value6, value7);
```

Because trace messages are usually needed only during debugging, whether trace messages are contained in a message file is left at the discretion of the developer:

- If non-English-speaking users need to view trace messages, you need to internationalize these messages. Therefore, you must put the trace messages in a message file and extract them with the `generateAndTraceMsg()` method. This message file should be the connector message file, which contains message specific to your connector.
- If only English-speaking users need to view trace messages, you do not need to internationalize these messages. Therefore, you can include the trace message (in English) directly in the call to `traceWrite()`. You do *not* need to use the `generateMsg()` or `generateAndTraceMsg()` method.

Connector messages logged with `generateAndTraceMsg()` are *not* viewable using LogViewer.

See also

generateAndLogMsg(),, generateMsg(),, traceWrite()

generateMsg()

Generates a message from a set of predefined messages in a message file.

Syntax

```
public final static String generateMsg(int traceLevel, int msgNum,
    int msgType, int isGlobal, int argCount,
    Vector msgParams);

public final static String generateMsg(int msgNum, int msgType,
    int isGlobal, int argCount, Vector msgParams);
```

Parameters

traceLevel Is one of the following trace-level constants defined in the `CWConnectorLogAndTrace` class to specify the trace level at which to generate the message:

```
CWConnectorLogAndTrace.LEVEL1
CWConnectorLogAndTrace.LEVEL2
CWConnectorLogAndTrace.LEVEL3
CWConnectorLogAndTrace.LEVEL4
CWConnectorLogAndTrace.LEVEL5
```

When this parameter is omitted, the method generates the message regardless of the trace level. The message is generated only if the *traceLevel* value is equal to or less than the current trace level of the connector.

msgNum Specifies the message number (identifier) in the message file.

msgType Is one of the following message-type constants defined in the `CWConnectorLogAndTrace` class to identify the message:

```
CWConnectorLogAndTrace.XRD_WARNING
CWConnectorLogAndTrace.XRD_ERROR
CWConnectorLogAndTrace.XRD_FATAL
CWConnectorLogAndTrace.XRD_INFO
CWConnectorLogAndTrace.XRD_TRACE
```

isGlobal Is the `CONNECTOR_MESSAGE_FILE` message-file constant defined in the `CWConnectorUtil` class to indicate that the message file is the connector message file.

argCount Is an integer that specifies the number of parameters within the message text. To determine the number, refer to the message in the message file.

msgParams Is a list of parameters for the message text.

Return values

A `String` containing the generated message. For the first form of the method, the method returns `null` if the trace level is greater than the trace level of the connector.

Exceptions

None.

Notes

The `generateMsg()` method provides two forms:

- Use the first form of the method (where *traceLevel* is the first parameter) for tracing messages. For the message to be generated, the trace level must be less than or equal to the trace level of the connector. You then use the `traceWrite()` method to send the trace message to the trace destination.

  You can use the `generateAndTraceMsg()` method to combine the message generation and tracing steps.
- Use the second form of the signature (where *msgNum* is the first parameter) for logging. You then use the `logMsg()` method to send the log message to the log destination.

  You can use the `generateAndLogMsg()` method to combine the message generation and logging steps.

See also

generateAndLogMsg(),, generateAndTraceMsg(),, logMsg(),, traceWrite()

getAllConfigProperties()

Retrieves a list of all configuration properties, as hierarchical connector properties, for the current connector.

Syntax

```
public static CWProperty[] getAllConfigProperties();
```

Parameters

None.

Return values

A reference to an array of `CWProperty` objects, each of which contains one connector property for the current connector.

Exceptions

None.

Notes
The `getAllConfigProperties()` method retrieves the connector configuration properties as an array of `CWProperty` objects. Each connector-property (`CWProperty`) object contains a single connector property and can hold a single value, another property, or a combination of values and child properties. Use methods of the `CWProperty` class (such as `getHierChildProps()` and `getStringValues()`) to obtain the values from a connector-property object.

See also
getConfigProp(),, getAllConnectorAgentProperties()

getAllConnectorAgentProperties()

Retrieves a list of all configuration properties for the current connector.

Syntax
```
public static Hashtable getAllConnectorAgentProperties();
```

Parameters
None.

Return values
A reference to a `java.util.Hashtable` object that contains the connector properties for the current connector.

Exceptions
None.

Notes
The `getAllConnectorAgentProperties()` method retrieves the connector configuration properties as a Java `Hashtable` object, which maps keys to values. The keys are the names of the properties and values are the associated property values. Use methods of the `Hashtable` class (such as `keys()` and `elements()`) to obtain the information from this structure.

Note: This method does *not* retrieve hierarchical or multi-valued properties. If it attempts to retrieve a multi-valued property, it returns only the first of the values. To retrieve hierarchical or multi-valued properties, use the `getAllConfigProperties()` method.

Examples
```
Hashtable ht = new Hastable();
ht = CWConnectorUtil.getAllConnectorAgentProperties();
int size = ht.size();
Enumeration properties = ht.keys();
Enumeration values = ht.elements();

while (properties.hasMoreElements()) {
    System.out.print((String)properties.nextElement());
    System.out.print("=");
    System.out.println((String)values.nextElement());
    System.out.println("Property set");
}
```

See also
getConfigProp(),, getAllConfigProperties()

getBlankValue()

Retrieves the value for the special Blank attribute value.

Syntax

```
public static String getBlankValue();
```

Return values

A `String` object containing the Blank attribute value.

Notes

The Blank value, which `getBlankValue()` retrieves, is a special attribute value that represents a "null" or zero-length value. Although the Java connector library does provide the `CWConnectorAttrType.CxBlank` constant for the Blank attribute value, it is recommended that you use the `getBlankValue()` method to obtain the Blank value when you want to assign it to an attribute.

See also

getIgnoreValue()

getConfigProp()

Retrieves the value of a connector configuration property.

Syntax

```
public final static String getConfigProp(String propName);
```

Parameters

propName Is the name of the property to retrieve.

Return values

A `String` object containing the property value. If the property name is not found, the method returns `null`.

Exceptions

None.

Notes

Values of connector configuration properties are downloaded to the connector during its initialization.

When you call `getConfigProp("ConnectorName")` in a parallel-process connector (one that has the `ParallelProcessDegree` connector property set to a value greater than 1), the method always returns the name of the connector-agent master process, regardless of whether it is called in the master process or a slave process.

See also

getAllConnectorAgentProperties(),, getHierarchicalConfigProp()

getCWConnectorAPIVersion()

Retrieves the version of the Java connector library.

Syntax

```
public static String getCWConnectorAPIVersion()
```

Parameters
None.

Return values
A `String` that contains the version of the Java connector library.

Exceptions
None.

Notes
The `getCWConnectorAPIVersion()` method retrieves the Java connector library version from the manifest file in the package. A manifest file is a standard Java file that stores version information for a product.

See also
traceCWConnectorAPIVersion()

getGlobalEncoding()

Retrieves the character encoding that the connector framework is using.

Syntax
```
public String getGlobalEncoding();
```

Parameters
None.

Return values
A `String` object containing the connector framework's character encoding.

Exceptions
None.

Notes
The `getGlobalEncoding()` method retrieves the connector framework's character encoding, which is part of the locale. The locale specifies cultural conventions for data according to language, country (or territory), and a character encoding. The connector framework's character encoding should indicate the character encoding of the connector application. The connector framework's character encoding using the following hierarchy:

- The `CharacterEncoding` connector configuration property in the repository

  WebSphere InterChange Server
  If a local configuration file exists, the setting of the `CharacterEncoding` connector configuration property in this local file takes precedence. If no local configuration file exists, the setting of the `CharacterEncoding` property is one from the set of connector configuration properties downloaded from the InterChange Server repository at connector startup.

- The character encoding from the Java environment, which Unicode (UCS-2)

This method is useful when the connector needs to perform character-encoding processing, such as character conversion.

See also
getGlobalLocale()

getGlobalLocale()

Retrieves the locale of the connector framework.

Syntax
```
public static String getGlobalLocale();
```

Parameters
None.

Return values
A `String` object containing the connector framework's locale setting.

Exceptions
None.

Notes
The `getGlobalLocale()` method retrieves the connector framework's locale, which defines cultural conventions for data according to language, country (or territory), and a character encoding. The connector framework's locale should indicate the locale of the connector application. The connector framework's locale is set using the following hierarchy:

- The `LOCALE` connector configuration property in the repository

> **WebSphere InterChange Server**
> If a local configuration file exists, the setting of the `Locale` connector configuration property in this local file takes precedence. If no local configuration file exists, the setting of the `Locale` property is the one from the set of connector configuration properties downloaded from the InterChange Server repository at connector startup.

- The locale from the Java environment, which is the locale from the operating system

This method is useful when the connector needs to perform locale-sensitive processing.

See also
createBusObj(),, getGlobalEncoding(),, getLocale()

getHierarchicalConfigProp()

Retrieves the top-level connector-object for a specified hierarchical connector configuration property.

Syntax
```
public static CWProperty getHierarchicalConfigProp(String propName);
```

Parameters
propName Is the name of the hierarchical connector property to retrieve.

Return values
A `CWProperty` object that contains the top-level connector-property object for the specified hierarchical connector property.

Exceptions
`WrongPropertyException`
Thrown if the specified connector-property name does not exist for this connector or it is not a hierarchical connector property.

Notes
The `getHierarchicalConfigProp()` method obtains the top-level connector-property (`CWProperty`) object. From this retrieved object, you can use methods of the `CWProperty` class to obtain the desired values of the connector property.

Note: Values of connector configuration properties are downloaded to the connector during its initialization. If you specify a *propName* for a connector property that has not been downloaded, `getHierarchicalConfigProp()` throws the `WrongPropertyException` exception.

When you call `getHierarchicalConfigProp("ConnectorName")` in a parallel-process connector (one that has the `ParallelProcessDegree` connector property set to a value greater than 1), the method always returns the name of the connector-agent master process, regardless of whether it is called in the master process or a slave process.

See also
getAllConfigProperties(),, getConfigProp()

getIgnoreValue()

Retrieves the value for the special Ignore attribute value.

Syntax
```
public static String getIgnoreValue();
```

Parameters
None.

Return values
A `String` object containing the Ignore attribute value.

Exceptions
None.

Notes
The Ignore value, which `getIgnoreValue()` retrieves, is a special attribute value that represents an attribute value that the connector can ignore. Although the Java connector library does provide the `CWConnectorAttrType.CxIgnore` constant for the Ignore attribute value, it is recommended that you use the `getIgnoreValue()` method to obtain the Ignore value when you want to assign it to an attribute.

See also
getBlankValue()

getSupportedBONames()

Retrieves a list of supported business objects for the current connector.

WebSphere InterChange Server

The `getSupportedBusObjNames()` method is valid *only* when the integration broker is InterChange Server (ICS). It can provide supported business objects only with ICS version 4.0 and later. For ICS versions earlier than 4.0, this method throws the `FunctionalityNotImplementedException` exception.

Syntax

```
public static String[] getSupportedBONames()
```

Parameters

None.

Return values

A `String` array that contains a list of the names of the supported business objects for the connector.

Exceptions

`NotSupportedException`
: Thrown if this method is called within a connector that has a version 3.x InterChange Server as its integration broker.

Notes

The `getSupportedBONames()` method returns a list of top-level supported business objects for the current connector; that is, if the connector supports business objects that contain child business objects, `getSupportedBONames()` includes only the name of the parent object in its list.

Note: The `getSupportedBONames()` method is *only* supported when the connector is using a version 4.0 or later InterChange Server as its integration broker. If the connector is using an earlier version of InterChange Server, the method returns the `NotSupportedException` exception.

getVersion()

Retrieves the version of the connector.

Syntax

```
public static String getVersion();
```

Return values

A `String` containing the version of the connector.

Exceptions

None.

Notes

The `getVersion()` method returns the version of the Java connector library. It obtains this version from the manifest file that is present in the Java package.

Note: The `CWConnectorAgent` class also provides a `getVersion()` method. However, this method retrieves the version of the connector itself.

initAndValidateAttributes()

Initializes attributes that do not have values set, but are marked as required, with their default values.

Syntax

```
public static final void initAndValidateAttributes(
   CWConnectorBusObj theBusObj) ;
```

Parameters

theBusObj Is the business object whose attributes this method sets.

Return values

None.

Exceptions

`SpecNameNotFoundException`
Thrown when the name of the specified business object does not match any of the business object definitions in the repository.

`DefaultSettingFailedException`
Thrown when the attribute's default value could not be set and there is no default value specified for the attribute in the business object definition.

Notes

The `initAndValidateAttributes()` method has two purposes:

- It *initializes* attributes by setting the value for each attribute to its default value under the following conditions:
  - When the `UseDefaults` connector configuration property is set to `true`
  - When the attribute's `isRequired` property is set to `true` (the attribute is required)
  - When the attribute's value is *not* currently set (has the special Ignore value of CxIgnore)
  - When the attribute's Default Value property specifies a default value
- It *validates* attributes by throwing a `DefaultSettingFailedException` exception under the following conditions:
  - When the attribute's `isRequired` property is set to `true`
  - When the attribute does *not* have a Default Value property that defines its default value

  In case of failure, no value exists some attributes (those without default values) after `initAndValidateAttributes()` finishes default-value processing. You might want to code your connector's application-specific component to catch this exception and return `CWConnectorConstant.FAIL`.

The `initAndValidateAttributes()` method looks at every attribute in all levels of a business object and determines the following:

- Whether an attribute is required
- Whether the attribute has a value in the business object instance
- Whether the `UseDefaults` configuration property is set to `true`

If an attribute is required and UseDefaults is true, initAndValidateAttributes() sets the value of any unset attribute to its default value. To have initAndValidateAttributes() set the attribute value to the special Blank value (CxBlank), you can set the attribute's default value to the string "CxBlank". If the attribute does *not* have a default value, initAndValidateAttributes() throws the DefaultSettingFailedException exception.

Note: If an attribute is a key or other attribute whose value is generated by the application, the business object definition should not provide default values, and the Required property for the attribute should be set to false.

The initAndValidateAttributes() method is usually called from the business-object-handler doVerbFor() method to ensure that required attributes have values before a Create operation is performed in an application. In the doVerbFor() method, you can call initAndValidateAttributes() for the Create verb. You can also call it for the Update verb, before it performs a Create.

To use initAndValidateAttributes(), you must also do the following:

- Design business object definitions so that the IsRequired attribute property is set to true for required attributes and that required attributes have default values specified in their Default Value property.
- Add the UseDefaults connector configuration property to the list of connector-specific properties for the connector. Set this property to true.

isBlankValue()

Determines if an attribute value is the special Blank value.

Syntax

```
public static boolean isBlankValue(Object value);
```

Parameters

value Is the value to compare with the special Blank value.

Return values

Returns true if the specified attribute value is the Blank attribute value; otherwise, returns false.

Exceptions

None.

See also

isIgnoreValue()

isIgnoreValue()

Determines if an attribute value is the special Ignore value.

Syntax

```
public static boolean isIgnoreValue(Object value);
```

Parameters

value Is the value to compare with the special Ignore value.

Return values
Return `true` if the specified attribute value is the Ignore value; otherwise, returns `false`.

Exceptions
None.

Notes
The Ignore attribute value signifies that this attribute is to be ignored while processing the business object.

See also
isBlankValue()

isTraceEnabled()

Determines if the trace level is greater than or equal to the trace level for which it is looking, if tracing is enabled at this level.

Syntax
```
public final static boolean isTraceEnabled(int traceLevel);
```

Parameters
traceLevel is the trace level to check.

Return values
Returns `true` if the agent trace level is greater than or equal to the trace level passed in.

Exceptions
None.

Notes
Use this method before generating a message.

logMsg()

Logs a message to the connector's log destination.

Syntax
```
public final static void logMsg(String msg, int severity);
```

Parameters
msg Is the message text to be logged.

severity Is one of the following message-type constants to identify the message:

```
CWConnectorUtil.XRD_WARNING
CWConnectorUtil.XRD_ERROR
CWConnectorUtil.XRD_FATAL
CWConnectorUtil.XRD_INFO
CWConnectorUtil.XRD_TRACE
```

Return values
None.

Exceptions
None.

Notes
The `logMsg()` method sends the specified *msg* text to the log destination. You establish the name of a connector's log destination through the Logging section in the Trace/Log File tab of Connector Configurator.

IBM recommends that log messages be contained in a message file and extracted with the `generateMsg()` method. This message file should be the connector message file, which contains messages specific to your connector. The `generateMsg()` method generates the message string for `logMsg()`. It retrieves a predefined message from a message file, formats the text, and returns a generated message string.

Note: You can use the `generateAndLogMsg()` method to combine the message generation and logging steps.

WebSphere InterChange Server

If `severity` is `XRD_ERROR` or `XRD_FATAL` and the connector configuration property `LogAtInterchangeEnd` is set, the error message is logged and an email notification is sent when email notification is on. See the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set for information on how to set up email notification for errors.

Connector messages logged with `logMsg()` are viewable using LogViewer if the message strings were generated with `generateMsg()`.

See also
generateAndLogMsg(),, generateMsg()

readerToBO()

Calls a data handler to convert serialized data of a specified MIME type to a business object. This serialized data is accessed as a `Reader` object.

Syntax
```
public static CWConnectorBusObj readerToBO(CWConnectorBusObj theBusObj,
      Reader serializedData, String mimeType, String BOPrefix,
      String encoding, Locale locale, Object config);
```

Parameters
BOPrefix Is the business-object prefix, which is combined with *mimeType* to form the key of the child meta-object. This argument can be used to specify a MIME subtype. It can also be used to specify a value for the `BOPrefix` data-handler configuration property.

config Is an `Object` that contains additional configuration information for the data handler.

encoding Specifies the character encoding for the serialized data in the `Reader` object. If you specify `null`, the method uses the character encoding of the machine.

locale Is a `java.util.Locale` object that specifies the locale for the serialized data in the `Reader` object. If you specify `null`, the method uses the connector-framework locale.

mimeType Is the MIME type that identifies the format of the serialized data.

serializedData Is an object of the Java `java.io.Reader` class (or one of its subclasses) that accesses the serialized data to convert to a business object.

theBusObj Identifies the type of business object (business object definition) to which the method converts the serialized data.

Return values

A `CWConnectorBusObj` object that contains the business object for the serialized data.

Exceptions

`DataHandlerCreateException`
Thrown when the `readerToBO()` method cannot instantiate a data handler for the specified MIME type.

`ParseException`
Thrown when the data handler encounters some error during the conversion of the serialized data to the specified business object.

`PropertyNotSetException`
Thrown when the `DataHandlerMetaObjectName` connector configuration property is not set.

The `readerToBO()` method can also throw the general Java exception `NullPointerException` if the data handler returns a `null` pointer instead of a business object.

Notes

The `readerToBO()` method provides the connector with the ability to call a data handler to perform string-to-business-object conversion. With this method, the incoming *serializedData* is accessed through a Java `Reader` object. The method identifies which data handler to invoke based on the specified *mimeType* argument. It instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object, as follows:

- It checks the top-level meta-object for the data handler that corresponds to this MIME type. It obtains the name of this top-level meta-object from the `DataHandlerMetaObjectName` connector configuration property. If this property is not set, `readerToBO()` throws the `PropertyNotSetException` exception.
- The instantiation process converts the specified *mimeType* to its equivalent MIME-type string and then searches the top-level meta-object for an attribute whose name matches this MIME-type string. The associated type for this attribute is the child meta-object.
- It obtains the name of the class to instantiate from the `ClassName` attribute in the child meta-object.

  If `readerToBO()` specifies a *BOPrefix* and a *mimeType*, it instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object. However, when a *BOPrefix* is specified, the instantiation process interprets this value as a MIME subtype. It searches the top-level meta-object for an attribute whose name includes *both* the MIME type and subtype.

If the data handler cannot be instantiated, `readerToBO()` throws the `DataHandlerCreateException`. For more information on how the arguments of `readerToBO()` identify which data handler to instantiate, see "Identifying the data handler to instantiate" on page 77.

Once instantiated, the data handler converts the specified serialized data to a business object of the type that *theBusObj* indicates. If `readerToBO()` specifies the *encoding* and *locale* arguments, the data handler uses the specified character encoding and locale when it interprets the serialized data. The data handler returns this business object to the `readerToBO()` method, which in turn returns it to the calling method.

Note: If the data handler cannot convert the serialized data, `readerToBO()` throws the `ParseException` exception.

You can specify a *config* option if you need to provide the data handler with more configuration information than is available in its meta-object. This argument can be used to specify a template file or a string to a URL for a schema that is used to build an XML document from a business object.

See also

byteArrayToBo(), streamToBO(), stringToBo()

streamToBO()

Calls a data handler to convert serialized data of a specified MIME type to a business object. This serialized data is accessed through an input stream.

Syntax

```
public static CWConnectorBusObj streamToBO(CWConnectorBusObj theBusObj,
      InputStream serializedData, String mimeType, String BOPrefix,
      String encoding, Locale locale, Object config);
```

Parameters

BOPrefix Is the business-object prefix, which is combined with *mimeType* to form the key of the child meta-object. This argument can be used to specify a MIME subtype. It can also be used to specify a value for the `BOPrefix` data-handler configuration property.

config Is an `Object` that contains additional configuration information for the data handler.

encoding Specifies the character encoding for the serialized data in the input stream. If you specify `null`, the method uses the character encoding of the machine.

locale Is a `java.util.Locale` object that specifies the locale for the serialized data in the input stream. If you specify `null`, the method uses the connector-framework locale.

mimeType Is the MIME type that identifies the format of the serialized data.

serializedData Is an object of the Java `java.io.InputStream` class (or one of its subclasses) that accesses the serialized data to convert to a business object.

theBusObj Identifies the type of business object (business object definition) to which the method converts the serialized data.

Return values

A CWConnectorBusObj object that contains the business object for the serialized data.

Exceptions

DataHandlerCreateException
: Thrown when the streamToBO() method cannot instantiate a data handler for the specified MIME type.

ParseException
: Thrown when the data handler encounters some error during the conversion of the serialized data to the specified business object.

PropertyNotSetException
: Thrown when the DataHandlerMetaObjectName connector configuration property is not set.

The streamToBO() method can also throw the general Java exception NullPointerException if the data handler returns a null pointer instead of a business object.

Notes

The streamToBO() method provides the connector with the ability to call a data handler to perform string-to-business-object conversion. With this method, the incoming *serializedData* is accessed through a Java input stream (derived from the InputStream class). The method identifies which data handler to invoke based on the specified *mimeType* argument. It instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object, as follows:

- It checks the top-level meta-object for the data handler that corresponds to this MIME type. It obtains the name of this top-level meta-object from the DataHandlerMetaObjectName connector configuration property. If this property is not set, streamToBO() throws the PropertyNotSetException exception.
- The instantiation process converts the specified *mimeType* to its equivalent MIME-type string and then searches the top-level meta-object for an attribute whose name matches this MIME-type string. The associated type for this attribute is the child meta-object.
- It obtains the name of the class to instantiate from the ClassName attribute in the child meta-object.

  If streamToBO() specifies a *BOPrefix* and a *mimeType*, it instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object. However, when a *BOPrefix* is specified, the instantiation process interprets this value as a MIME subtype. It searches the top-level meta-object for an attribute whose name includes *both* the MIME type and subtype.

If the data handler cannot be instantiated, streamToBO() throws the DataHandlerCreateException. For more information on how the arguments of streamToBO() identify which data handler to instantiate, see "Identifying the data handler to instantiate" on page 77.

Once instantiated, the data handler converts the specified serialized data to a business object of the type that *theBusObj* indicates. If streamToBO() specifies the *encoding* and *locale* arguments, the data handler uses the specified character encoding and locale when it interprets the serialized data. The data handler returns this business object to the streamToBO() method, which in turn returns it to the calling method.

Note: If the data handler cannot convert the serialized data, `streamToBO()` throws the `ParseException` exception.

You can specify a *config* option if you need to provide the data handler with more configuration information than is available in its meta-object. This argument can be used to specify a template file or a string to a URL for a schema that is used to build an XML document from a business object.

See also

boToStream(), byteArrayToBo(), readerToBO(), stringToBo()

stringToBo()

Calls a data handler to convert serialized data of a specified MIME type to a business object. This serialized data is accessed as a string.

Syntax

```
public static CWConnectorBusObj stringToBo(CWConnectorBusObj theBusObj,
      String serializedData,     String mimeType);
public static CWConnectorBusObj stringToBo(CWConnectorBusObj theBusObj,
      String serializedData, String mimeType, Object config);
public static CWConnectorBusObj stringToBo(CWConnectorBusObj theBusObj,
      String serializedData, String mimeType, String BOPrefix,
      String encoding, Locale locale, Object config);
```

Parameters

BOPrefix Is the business-object prefix, which is combined with *mimeType* to form the key of the child meta-object. This argument can be used to specify a MIME subtype. It can also be used to specify a value for the `BOPrefix` data-handler configuration property.

config Is an `Object` that contains additional configuration information for the data handler.

encoding Specifies the character encoding for the serialized data in the `String`. If you specify `null`, the method uses the character encoding of the machine.

locale Is a `java.util.Locale` object that specifies the locale for the serialized data in the `String`. If you specify `null`, the method uses the connector-framework locale.

mimeType Is the MIME type that identifies the format of the serialized data.

serializedData Is the string representation of the serialized data to convert to a business object.

theBusObj Identifies the type of business object (business object definition) to which the method converts the serialized data.

Return values

A `CWConnectorBusObj` object that contains the business object for the serialized data.

Exceptions

`DataHandlerCreateException`
Thrown when the `stringToBo()` method cannot instantiate a data handler for the specified MIME type.

`ParseException`
: Thrown when the data handler encounters some error during the conversion of the serialized data to the specified business object.

`PropertyNotSetException`
: Thrown when the `DataHandlerMetaObjectName` connector configuration property is not set.

The `stringToBo()` method can also throw the general Java exception `NullPointerException` if the data handler returns a `null` pointer instead of a business object.

Notes

The `stringToBo()` method provides the connector with the ability to call a data handler to perform string-to-business-object conversion. With this method, the incoming *serializedData* is accessed through a Java `String`. The method identifies which data handler to invoke based on the specified *mimeType* argument. It instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object, as follows:

- It checks the top-level meta-object for the data handler that corresponds to this MIME type. It obtains the name of this top-level meta-object from the `DataHandlerMetaObjectName` connector configuration property. If this property is not set, `stringToBo()` throws the `PropertyNotSetException` exception.
- The instantiation process converts the specified *mimeType* to its equivalent MIME-type string and then searches the top-level meta-object for an attribute whose name matches this MIME-type string. The associated type for this attribute is the child meta-object.
- It obtains the name of the class to instantiate from the `ClassName` attribute in the child meta-object.

  If `stringToBo()` specifies a *BOPrefix* and a *mimeType*, it instantiates a data handler whose class name is identified in the child meta-object associated with this MIME type in the the top-level meta-object. However, when a *BOPrefix* is specified, the instantiation process interprets this value as a MIME subtype. It searches the top-level meta-object for an attribute whose name includes *both* the MIME type and subtype.

If the data handler *cannot* be instantiated, `stringToBo()` throws the `DataHandlerCreateException`. For more information on how the arguments of `stringToBo()` identify which data handler to instantiate, see "Identifying the data handler to instantiate" on page 77.

Once instantiated, the data handler converts the specified serialized data to a business object of the type that *theBusObj* indicates. If `stringToBo()` specifies the *encoding* and *locale* arguments, the data handler uses the specified character encoding and locale when it interprets the serialized data. The data handler returns this business object to the `stringToBo()` method, which in turn returns it to the calling method.

Note: If the data handler cannot convert the serialized data, `stringToBo()` throws the `ParseException` exception.

You can specify a *config* option if you need to provide the data handler with more configuration information than is available in its meta-object. This argument can be used to specify a template file or a string to a URL for a schema that is used to build an XML document from a business object.

See also
boToString(), byteArrayToBo(), readerToBO(), streamToBO()

traceCWConnectorAPIVersion()

Writes the version of the Java connector library to the trace destination.

Syntax
```
public static void traceCWConnectorAPIVersion();
```

Parameters
None.

Return values
None.

Exceptions
None.

Notes
The `traceCWConnectorAPIVersion()` method sends the version of the Java connector library to the trace destination when the trace level is at level 1 and higher. It obtains the version of Java connector library from the manifest file in the package. A manifest file is a standard Java file that stores version information for a product.

You establish the name of a connector's trace destination through the `TraceFileName` connector configuration property.

See also
getCWConnectorAPIVersion()

traceWrite()

Writes a trace message to the trace destination.

Syntax
```
public final static void traceWrite(int traceLevel, String msg);
```

Parameters

traceLevel Is one of the following trace-level constants to identify the trace level used to determine which trace messages are output:

```
CWConnectorUtil.LEVEL1
CWConnectorUtil.LEVEL2
CWConnectorUtil.LEVEL3
CWConnectorUtil.LEVEL4
CWConnectorUtil.LEVEL5
```

The method writes the trace message when the current trace level is greater than or equal to *traceLevel*.

Note: Do not specify a trace level of zero (`LEVEL0`) with a tracing message. A trace level of zero indicates that tracing is turned off. Therefore, any trace message associated with a *traceLevel* of `LEVEL0` will never print.

msg Is the message text to use for the trace message.

Return values
None.

Exceptions
None.

Notes
You can use the `traceWrite()` method to write your own trace messages for a connector. Tracing is turned on for the connector when the `TraceLevel` connector configuration property is set to a nonzero value (any trace-level constant *except* `LEVEL0`).

The `traceWrite()` method sends the specified *msg* text to the trace destination when the current trace level is greater than or equal to *traceLevel*. You establish the name of a connector's trace destination through the Tracing section in the Trace/Log File tab of Connector Configurator.

Because trace messages are usually needed only during debugging, whether trace messages are contained in a message file is left at the discretion of the developer:

- If non-English-speaking users need to view trace messages, you need to internationalize these messages. Therefore, you must put the trace messages in a message file and extract them with the `generateMsg()` method. This message file should be the connector message file, which contains message specific to your connector. The `generateMsg()` method generates the message string for `traceWrite()`. It retrieves a predefined trace message from a message file, formats the text, and returns a generated message string.

  Note: You can use the `generateAndTraceMsg()` method to combine the message generation and logging steps.
- If only English-speaking users need to view trace messages, you do not need to internationalize these messages. Therefore, you can include the trace message (in English) directly in the call to `traceWrite()`. You do *not* need to use the `generateMsg()` or `generateAndTraceMsg()` method.

Connector messages logged with `traceWrite()` are *not* viewable using LogViewer.

See also
generateAndTraceMsg(),, generateMsg()

Deprecated Methods

Some methods in the `CWConnectorUtil` class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but IBM recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 139 lists the deprecated methods for the `CWConnectorUtil` class. If you are writing a new connector (not modifying an existing connector), you can ignore this section.

Table 139. Deprecated methods of the CWConnectorUtil class

Deprecated method	Replacement
`generateAndLogMsg()` Fourth argument was a count of the number of arguments (*`argCount`*) in the *`msgParameters`* list.	generateAndLogMsg() The *`argCount`* value is no longer required for this method and can be omitted. The method itself can determine the number of arguments in the *`msgParameters`* list.
`generateAndTraceMsg()` Fifth argument was a count of the number of arguments (*`argCount`*) in the *`msgParameters`* list.	generateAndTraceMsg() The *`argCount`* value is no longer required for this method and can be omitted. The method itself can determine the number of arguments in the *`msgParameters`* list.
`generateAndTraceMsg()` Third argument was the message type of the message to generate (*`msgType`*).	generateAndTraceMsg() The *`msgType`* value is no longer required for this method and can be omitted. Because the message type for a trace message should always be `XRD_TRACE`, the method itself can fill in the message type.

Chapter 23. CWCustomBOHandlerInterface interface

The CWCustomBOHandlerInterface interface defines the behavior of a custom business object handler. It provides the code to implement and access one business object handler. Normally, you provide a business object handler by extending the business-object-handler base class, CWConnectorBOHandler and implementing the doVerbFor() method to define the business object handler's functionality. With this approach, there is a standard way of handling each verb for a business object. However, this mechanism does *not* support the ability to customize the behavior of a particular verb for only some business objects.

If you need to provide customized behavior for a business object handler, you can create a custom business object handler by creating a custom-business-object-handler class and implementing the CWCustomBOHandlerInterface interface. Its doVerbForCustom() method defines the functionality for this custom business-object handler.

For an introduction to request processing and business object handlers, see "Request processing" on page 24. For information on how to implement a business object handler, see Chapter 4, "Request processing," on page 79.

Table 140 summarizes the methods in the CWCustomBOHandlerInterface interface.

Table 140. Member method of the CWCustomBOHandlerInterface interface

Member method	Description	Page
doVerbForCustom()	Performs the verb processing for the active verb of a business object.	381

doVerbForCustom()

Performs the custom verb processing for the active verb of a business object.

Syntax

```
public int doVerbForCustom(CWConnectorBusObj theBusObj);
```

Parameters

theBusObj Is the business object whose active verb is to be processed.

Return values

An integer that indicates the outcome status of the verb operation. Compare this integer value with the following outcome-status constants to determine the status:

CWConnectorConstant.SUCCEED
: The verb operation succeeded.

CWConnectorConstant.FAIL
: The verb operation failed.

CWConnectorConstant.APPRESPONSETIMEOUT
: The application is not responding.

`CWConnectorConstant.VALCHANGE`
: At least one value in the business object changed.

`CWConnectorConstant.VALDUPES`
: The requested operation found multiple records in the application database for the same key value.

`CWConnectorConstant.MULTIPLE_HITS`
: The connector finds multiple matching records when retrieving using non-key values. The connector returns a business object only for the first matching record.

`CWConnectorConstant.RETRIEVEBYCONTENT_FAILED`
: The connector was not able to find matches for Retrieve by non-key values.

`CWConnectorConstant.BO_DOES_NOT_EXIST`
: The connector performed a Retrieve operation, but the application database does not contain a matching entity for the requested business object.

Exceptions

`ConnectionFailureException`
: Thrown if the connector has lost the connection with the application.

`VerbProcessingFailedException`
: Thrown if the verb processing fails.

Notes

The `doVerbForCustom()` method performs the action of the active verb in the *theBusObj* business object, if this business object's verb application-specific information contains the `CBOH` tag. This tag specifies the entire class name (including its package name) for your implementation of the `CWCustomBOHandlerInterface` interface. For information on the format of this tag, see "Adding the verb application-specific information" on page 176.

When a business object handler is invoked, the low-level `doVerbFor()` method, (inherited from the `BOHandlerBase` class) is what the connector framework actually invokes. The low-level `doVerbFor()` method determines which business object handler to call as follows:

- If the business object's verb has the `CBOH` tag in its application-specific information, call this `doVerbForCustom()` method.
- Otherwise, call the `doVerbFor()` method, which the connector developer must implement as part of the business object handler's `CWConnectorBOHandler` class

For more information, see "Populating the return-status descriptor" on page 170.

If the `doVerbForCustom()` method needs to throw one of its exceptions, it first needs to populate an exception-detail object that it contain information about the exception. For more information, see Table 121 on page 253. For information about how to implement this method, see "Implementing the doVerbForCustom() method" on page 175.

See also

doVerbFor()

Chapter 24. CWException class

The CWException class is the base class for exceptions in the Java connector library. The Java connector library extends the Java Exception class to create its own exception class called:

```
com.crossworlds.cwconnectorapi.exceptions.CWException
```

This class represents an *exception object*, which methods of the Java connector library can throw.

Note: The reference description for most Java connector library methods lists the exceptions thrown by that method in the Exceptions section.

The CWException class provides the following:

- "Methods"
- "Exception subclasses" on page 386

Methods

Table 141 summarizes the methods in the CWException class.

Table 141. Member methods of the CWException class

Member method	Description	Page
CWException()	Creates an exception object.	383
getExceptionObject()	Retrieves an exception-detail object from the exception object.	384
getMessage()	Retrieves the message from the exception object.	384
getStatus()	Retrieves the status associated with the exception object.	385
setStatus()	Sets the status associated with the exception object.	385

CWException()

Creates an exception object.

Syntax

```
public CWException();
public CWException(CWExceptionObject excptionDetail);
```

Parameters

excptionDetail Is an exception-detail object that contains the additional exception information.

Return values

A new CWException object.

Notes

The `CWException()` constructor provides two forms:

- The first form creates an empty `CWException` object.
- The second form passes an exception-detail object to initialize the new `CWException` object.

getExceptionObject()

Retrieves an exception-detail object from the exception object.

Syntax

```
public CWConnectorExceptionObject getExceptionObject();
```

Parameters

None.

Return values

A `CWConnectorExceptionObject` object that contains the additional exception information.

Exceptions

None.

Notes

The `getExceptionObject()` method retrieves exception-detail information, in the form of a `CWConnectorExceptionObject`, from the exception object. You can use methods of the `CWConnectorExceptionObject` class to obtain exception information such as the message text, message number, and message explanation.

See also

Chapter 19, "CWConnectorExceptionObject class"

getMessage()

Retrieves the message from the exception object.

Syntax

```
public String getMessage();
```

Parameters

None.

Return values

A `String` object that contains the message associated with the exception.

Exceptions

None.

getStatus()

Retrieves the status associated with the exception object.

Syntax

```
public int getStatus();
```

Parameters

None.

Return values

The integer exception status in the exception object.

Exceptions

None.

Notes

The `getStatus()` method retrieves the status that is set by the connector. This status is usually one of the outcome-status constants, as represented by the `CWConnectorConstant` class (such as `FAIL` or `APPRESPONSETIMEOUT`).

See also

setStatus()

setStatus()

Sets the status associated with the exception object.

Syntax

```
public void setStatus(int status);
```

Parameters

status Is the integer status value to assign to the exception object.

Return values

None.

Exceptions

None.

Notes

The `setStatus()` method sets the status that is part of the `CWException` object. This status is usually set by the connector to one of the outcome-status constants, as represented by the `CWConnectorConstant` class (such as `FAIL` or `APPRESPONSETIMEOUT`).

See also

getStatus()

Exception subclasses

Within this CWException class are subclasses that identify particular exceptions possible in the methods of the Java connector library. Table 142 lists the subclassed exceptions.

Table 142. CWConnectorException subclasses

Exception subclass	Definition
ArchiveFailedException	Thrown from the archiveEvent() method of the event-store class if the event record could not be archived into the archive store.
AttributeNotFoundException	Thrown when the specified position or name of an attribute does not match the attribute name or attribute position within the existing business object.
AttributeNullValueException	Thrown if the attribute value is null when some operations need to be performed on the attribute value.
AttributeValueException	Thrown if there is a NumberFormatException exception.
ConnectionFailureException	Thrown if the connector is unable to establish a connection with the application.
DataHandlerCreateException	Thrown when a data-handler method cannot instantiate a data handler for the specified MIME type.
DefaultSettingFailedException	Thrown when setting a default value fails.
DeleteFailedException	Thrown from the deleteEvent() method of the event-store class if the event record could not be deleted from the event store.
InProgressEventRecoveryFailedException	Thrown if the recovery of the In-Progress events fails.
InvalidAttributePropertyException	Thrown when any invalid property of the attribute is queried (such as calling getMaxLength() on an attribute that is an object).
InvalidStatusChangeException	Thrown if the requested change in event status is not valid.
InvalidVerbException	Thrown when the specified verb is not supported by the business object.
LogonFailedException	Thrown if the connector is not able to logon to the application with the user name and password provided.
NotSupportedException	Thrown if some feature is not supported by the current version of the produce.
ParseException	Thrown when the data handler (called from the connector) encounters some error during conversion between the business object and the specified MIME type.
PropertyNotSetException	Thrown if a required connector configuration property is not set.
SpecNameNotFoundException	Thrown when the business object definition for creating a business object cannot be found.
StatusChangeFailedException	Thrown if the connector is not able to set the status of an event in the application's event store.
VerbProcessingFailedException	Thrown from the doVerbFor() method if the operation specified by the verb fails.
WrongASIFormatException	Thrown if the application-specific information is not in the format: *name=value*
WrongAttributeException	Thrown when the data type of the specified attribute does not match the data type that the attribute is defined to hold.

Table 143. Methods that return exceptions

Java connector library exception	Method that returns the exception
SpecNameNotFoundException	CWConnectorUtil createBusObj() CWConnectorBusObj setBusObjValue()
AttributeNotFoundException	CWConnectorBusObj getAttrIndex() getbooleanValue() getBusObjValue() getCardinality() getDefault() getDefaultboolean() getDefaultdouble() getDefaultfloat() getDefaultint() getDefaultlong() getDefaultString() getdoubleValue() getfloatValue() getintValue() getlongValue() getMaxLength() getObjectCount() getStringValue() getTypeName() getTypeNum() hasCardinality() hasName() hasType() isForeignKeyAttr() isKeyAttr() isMultipleCard() isObjectType() isRequiredAttr() isType() removeAllObjects() removeBusinessObjectAt() setbooleanValue() setBusObjValue() setdoubleValue() setfloatValue() setintValue() setStringValue()
WrongAttributeException	CWConnectorBusObj getbooleanValue() getBusObjValue() getDefaultboolean() getDefaultdouble() getDefaultfloat() getDefaultint() getDefaultlong() getDefaultString() getdoubleValue() getfloatValue()

Table 143. Methods that return exceptions (continued)

Java connector library exception	Method that returns the exception
	getintValue()
	getlongValue()
	getStringValue()
	setbooleanValue()
	setBusObjValue()
	setdoubleValue()
	setfloatValue()
	setintValue()
	setStringValue()
`AttributeNullValueException`	`CWConnectorBusObj`
	`getbooleanValue()`
	getDefaultboolean()
	getDefaultdouble()
	getDefaultfloat()
	getDefaultint()
	getDefaultlong()
	getdoubleValue()
	getfloatValue()
	getintValue()
	getlongValue()
	setBusObjValue()
`AttributeValueException`	`CWConnectorBusObj`
	getDefaultdouble()
	getDefaultfloat()
	getDefaultint()
	getDefaultlong()
	getdoubleValue()
	getfloatValue()
	getintValue()
	getlongValue()
	setbooleanValue()
	setBusObjValue()
	setdoubleValue()
	setfloatValue()
	setintValue()
	setStringValue()
`InvalidAttributePropertyException`	CWConnectorBusObj
	getMaxLength()
`InvalidVerbException`	CWConnectorBusObj
	setVerb()

Exception subclass constructor

Creates an exception subclass.

Syntax

`public` *`exception_subclass`*`(CWConnectorExceptionObject` *`excption`*`)`

where `exception_subclass` is the name of the exception subclass (as shown in Table 142).

Parameters

excption is an exception object that contains information about the exception.

Return values

An object that represents a subclass of the `CWException` class.

Notes

Use methods of the `CWConnectorExceptionObject` class to obtain information about the exception.

Chapter 25. CWProperty class

The CWProperty class represents a hierarchical connector configuration property for a Java connector. A hierarchical connector configuration property can contain one or more values and these values can be either string values or other (child) connector properties.

Note: The CWProperty class extends the CxProperty class of the low-level Java connector library. For more information on the classes of the low-level Java connector library, see Chapter 26, "Overview of the low-level Java connector library," on page 405.

Table 144 summarizes the methods in the CWProperty class.

Table 144. Member methods of the CWProperty class

Member method	Description	Page
CWProperty()	Creates a connector-property object.	391
getCardinality()	Retrieves the cardinality of the connector configuration property (single-valued or multi-values).	392
getChildPropValue()	Retrieves all string values for a specified child property.	393
getChildPropsWithPrefix()	Retrieves all child properties from the hierarchical connector configuration property whose names match a specified prefix.	393
getEncryptionFlag()	Retrieves the encryption flag for the connector configuration property.	394
getHierChildProp()	Retrieves *a specified* child property from the hierarchical connector configuration property.	395
getHierChildProps()	Retrieves *all* child properties from the hierarchical connector configuration property.	396
getHierProp()	Retrieves *a specified* child property from the hierarchical connector configuration property., at any level in the property hierarchy.	397
getName()	Retrieves the name of the connector configuration property.	398
getPropType()	Retrieves the property type for the connector configuration property (simple or hierarchical).	398
getStringValues()	Retrieves all string values from the hierarchical connector configuration property.	398
hasChildren()	Determines whether the connector configuration property has any child properties.	399
hasValue()	Determines whether the connector configuration property has any string values.	400
setEncryptionFlag()	Sets the encryption flag for the hierarchical connector configuration property.	401
setValues()	Sets the values of the hierarchical connector configuration property.	401

CWProperty()

Creates a hierarchical connector-property object.

Syntax

```
public CWProperty();
public CWProperty(String propName, String simplePropValue);
public CWProperty(String propName, CWProperty[] hierPropValues);
```

Parameters

propName Specifies the name of the connector configuration property.

simplePropValue
Is a String value with which to initialize a simple connector property.

hierPropValues Is an array of connector-property (`CWProperty`) objects with which to initialize a hierarchical connector property.

Return values

A `CWProperty` object containing the newly created hierarchical connector property.

Notes

The `CWProperty()` constructor provides the following forms:

- The first form creates an empty connector-property object. You can use other methods of the `CWProperty` class to populate this object.
- The second form creates a connector-property object for a simple connector property, with a property name and a string value that you specify.
- The third form creates a connector-property object for a hierarchical connector property, with a property name and array of hierarchical properties that you specify.

---

getCardinality()

Retrieves the cardinality of the connector configuration property.

Syntax

```
public int getCardinality();
```

Parameters

None.

Return values

An integer that indicates the cardinality of the connector configuration property. Compare this integer value with the following connector-property constants to determine the cardinality:

`CWConnectorConstant.SINGLE_VALUED`
The connector configuration property has single cardinality; that is, it contains only one value.

`CWConnectorConstant.MULTI_VALUED`
The connector configuration property has multiple cardinality; that is, it contains more than one value.

Exceptions

None.

Notes

The `getCardinality()` method retrieves the *cardinality* of a connector configuration property, which indicates whether the property contains one or many values. Use this method to determine how to retrieve the property values:

getChildPropValue()

Retrieves the string values from a specified child property in the hierarchical connector property.

Syntax

```
public String[] getChildPropValue(String propName);
```

Parameters

propName Specifies the name of the connector configuration property whose string values are retrieved.

Return values

A reference to an array of `String` objects, each of which represents one string value for the specified child property. If the specified child property does not exist in the current hierarchical connector property, the method returns `null`.

Exceptions

None.

Notes

The `getChildPropValue()` retrieves a string values for a specified child property. Before a call to `getChildPropValue()`, you can use the `hasValue()` method to verify that the hierarchical connector property has string values. To retrieve all string values of a hierarchical connector property, use the `getStringValues()` method.

If a hierarchical connector property has encrypted string values (its encrypted flag is `true`), the `getChildPropValue()` returns the unencrypted values. You do *not* have to handle decryption.

See also

getStringValues(),, hasValue()

getChildPropsWithPrefix()

Retrieves *all* child properties for the hierarchical connector configuration property whose names match a specified prefix.

Syntax

```
public CWProperty[] getChildPropsWithPrefix(String propPrefix);
```

Parameters

propPrefix Specifies the prefix to match in searching for child properties of the hierarchical connector configuration property.

Return values

A reference to an array of `CWProperty` objects, each of which represents one connector property in the hierarchical connector property whose name begins with the specified *propPrefix*. If no child properties exist in the hierarchical connector property with the specified prefix, the method returns `null`.

Exceptions

None.

Notes

The `getChildPropsWithPrefix()` method retrieves *all* child properties for the hierarchical connector configuration property whose name begins with the specified *propPrefix*. The retrieved properties are only those of the children of the current hierarchical property; they do *not* include any grandchildren, great-grandchildren, and so on. To retrieve child properties at lower levels in the hierarchy, you must first obtain the connector-property object for a property at a particular level and then use a method such as `getHierChildProps()` or `getHierChildProp()` to retrieve its children.

Note: You can use the `getHierProp()` to retrieve a specified child, grandchild, and so on down the property hierarchy.

For example, suppose you configure properties for multiple listeners with the following property hierarchy shown in Figure 77.

```
ProtocolListener
    SingleValProp1=dexter
    Listener1=first listener
        Port=1500
    Listener2=second listener
        Port=1502
    SingleValProp2=tashi
```

Figure 77. Sample property hierarchy for protocol listeners

To obtain all the properties with the prefix of "Listener", you must first retrieve the top-level connector-object for `ProtocolListener` (for example, into `topLevelProp`). You can then use the following call to retrieve both the `Listener1` and `Listener2` child properties of `ProtocolListener`:

```
CWProperty[] listenerProps = topLevelProp.getChildPropsWithPrefix("Listener");
```

Before a call to `getChildPropsWithPrefix()`, you can use the `hasChildren()` method to verify that the hierarchical connector property has child properties. To retrieve a specified child property, use the `getHierChildProp()` method. To retrieve all child properties, regardless of prefix, you can use the `getHierChildProps()` method.

See also

getHierChildProp(),, getHierChildProps(),, getHierProp(),, hasChildren()

---

getEncryptionFlag()

Retrieves the encryption flag of the hierarchical connector configuration property from the connector-property object.

Syntax

```
public Boolean getEncryptionFlag();
```

Parameters

None.

Return values

A `boolean` value that indicates whether the current connector configuration property's value is encrypted.

Exceptions

None.

Notes

The `getEncryptionFlag()` method obtains the boolean encryption flag from the connector-property object. This flag indicates whether the connector property's string values are encrypted.

Note: In Connector Configurator, encrypted values display as a string of asterisk (*) characters.

See also

setEncryptionFlag()

---

getHierChildProp()

Retrieves a specified child property for the hierarchical connector configuration property.

Syntax

```
public CWProperty getHierChildProp(String propName);
```

Parameters

propName Specifies the name of the connector configuration property to retrieve.

Return values

A `CWProperty` object that contains the retrieved child property. If the specified property does not exist in the current hierarchical connector property, the method returns `null`.

Exceptions

None.

Notes

The `getHierChildProp()` method retrieves the child property whose name matches *propName* from the hierarchical connector configuration property. The retrieved property must exist as a child of the current hierarchical property; it cannot be a grandchildren, great-grandchild, and so on. To retrieve child properties at lower

levels in the hierarchy, you must first obtain the connector-property object for a property at a particular level and then use a method such as `getHierChildProps()` or `getHierChildProp()` to retrieve its children.

Note: You can use the `getHierProp()` to retrieve a specified child, grandchild, and so on down the property hierarchy.

Before a call to `getHierChildProp()`, you can use the `hasChildren()` method to verify that the hierarchical connector property has child properties. To retrieve *all* child properties, use the `getHierChildProps()` method. To retrieve all child properties with a specified prefix, you can use the `getChildPropsWithPrefix()` method.

See also

getChildPropsWithPrefix(),, getHierChildProps(),, getHierProp(),, hasChildren(),, setValues()

---

getHierChildProps()

Retrieves *all* child properties for the hierarchical connector configuration property.

Syntax

```
public CWProperty[] getHierChildProps();
```

Parameters

None.

Return values

A reference to an array of `CWProperty` objects, each of which represents one connector property in the hierarchical connector property. If the hierarchical connector property does not contain any child properties, the method returns `null`.

Exceptions

None.

Notes

The `getHierChildProps()` method retrieves *all* child properties for the hierarchical connector configuration property. The retrieved properties are only those of the children of the current hierarchical property; they do *not* include any grandchildren, great-grandchildren, and so on. To retrieve child properties at lower levels in the hierarchy, you must first obtain the connector-property object for a property at a particular level and then use a method such as `getHierChildProps()` or `getHierChildProp()` to retrieve its children.

Note: You can use the `getHierProp()` to retrieve a specified child, grandchild, and so on down the property hierarchy.

Before a call to `getHierChildProps()`, you can use the `hasChildren()` method to verify that the hierarchical connector property has child properties. To retrieve a specified child property, use the `getHierChildProp()` method. To retrieve all child properties with a specified prefix, you can use the `getChildPropsWithPrefix()` method. To retrieve all string values, use the `getStringValues()` method.

See also

getChildPropsWithPrefix(),, getHierChildProp(),, getHierProp(),, getStringValues(),, hasChildren(),, setValues()

---

getHierProp()

Retrieves a specified child property for the hierarchical connector configuration property *at any level* of the property hierarchy.

Syntax

```
public CWProperty getHierProp(String propName);
```

Parameters

propName Specifies the name of the connector configuration property to retrieve.

Return values

A `CWProperty` object that contains the retrieved property from the hierarchy. If the specified property does not exist in the current hierarchical connector property, the method returns `null`.

Exceptions

None.

Notes

The `getHierProp()` method retrieves the child property whose name matches *propName* from the hierarchical connector configuration property. You can retrieve a child property *at any level* of the current property hierarchical; you can specify a grandchild, great-grandchild, and so on. The *propName* of the retrieved child property has the form:

child/grandchild/great-grandchild/....

For example, suppose you have the property hierarchy shown in Figure 77 on page 394. To obtain the name of the port for `Listener1`, you must first retrieve the top-level connector-object for `ProtocolListener` (for example, into `topLevelProp`). You can then use the following call to retrieve the port name of `Listener1`:

```
CWProperty listenerPort = topLevelProp.getHierProp("Listener1/Port");
```

Before a call to `getHierProp()`, you can use the `hasChildren()` method to verify that the hierarchical connector property has child properties. To retrieve a specified child property at the top level of the property hierarchy, use the `getHierChildProp()` method. To retrieve *all* child properties at the top level of the hierarchy, you can use the `getHierChildProps()` method.

See also

getHierChildProp(),, getHierChildProps(),, hasChildren()

getName()

Retrieves the name of the hierarchical connector configuration property from the connector-property object.

Syntax

```
public String getName();
```

Parameters

None.

Return values

A String that contains the name of the connector configuration property.

Exceptions

None.

getPropType()

Retrieves the property type from a connector-property object.

Syntax

```
public int getPropType();
```

Parameters

None.

Return values

An integer that indicates the property type of the connector configuration property. Compare this integer value with the following connector-property constants to determine the type:

`CWConnectorConstant.SIMPLE`
: The connector configuration property is *simple*; that is, it contains *only* string values.

`CWConnectorConstant.HIERARCHICAL`
: The connector configuration property is *hierarchical*; that is, it contains one or more child properties and perhaps string values as well.

Exceptions

None.

getStringValues()

Retrieves *all* string values for the hierarchical connector configuration property.

Syntax

```
public String[] getStringValues();
```

Parameters

None.

Return values

A reference to an array of `String` objects, each of which represents one string value for the hierarchical connector property. If the hierarchical connector property does not contain any string values, the method returns `null`.

Exceptions

None.

Notes

The `getStringValues()` method retrieves *all* string values for the hierarchical connector configuration property. The retrieved string values are only those of the current hierarchical property; they do *not* include any values in child properties. To retrieve string values at lower levels in the hierarchy, you can do either of the following:

- Use the `getChildPropValue()` method to retrieve the string values of a specified child property.
- Obtain the connector-property object for a property at a particular level and then use a method such as `getStringValues()` to retrieve its string values.

Before a call to `getStringValues()`, you can use the `hasValue()` method to determine if the hierarchical connector property has any string values. To retrieve child properties, use the `getHierChildProp()` or `getHierChildProps()` method.

If a hierarchical connector property has encrypted string values (its encrypted flag is `true`), the `getStringValues()` returns the unencrypted values. You do *not* have to handle decryption.

See also

getChildPropValue(),, getHierChildProp(),, getHierChildProps(),, hasValue(),, setValues()

---

hasChildren()

Determines whether the current connector property contains any child properties.

Syntax

```
public boolean hasChildren();
```

Parameters

None.

Return values

A `boolean` that indicates whether the hierarchical connector property contains any child properties. The method returns `true` if it does contain child properties; otherwise, it returns `false`.

Exceptions

None.

Notes

The `hasChildren()` method is useful for determining which of the `CWProperty` methods to use to extract the value of a hierarchical connector property:

- If `hasChildren()` returns `true`, use one of the following value methods to retrieve child properties:

To obtain all child properties	`getHierChildProps()`
To obtain a specified child property	`getHierChildProp()`
To obtain all child properties whose names begin with a specified prefice	`getChildPropsWithPrefix()`

- If `hasChildren()` returns `false`, use one of the following value methods to retrieve any string values:

To obtain all string values	`getStringValues()`
To obtain the string values of a specified child property	`getChildPropValue()`

See also

getChildPropValue(),, getHierChildProp(),, getHierChildProps(),, getStringValues(),, hasValue()

---

hasValue()

Determines whether the current hierarhical connector property has any string values.

Syntax

```
public boolean hasValue();
```

Parameters

None.

Return values

A `boolean` that indicates whether the connector property contains any string values. The method returns `true` if it does contain values; otherwise, it returns `false`.

Exceptions

None.

Notes

The `hasValue()` method is useful for determining which of the `CWProperty` methods to use to extract the value of a hierarchical connector property:

- If `hasValue()` returns `true`, use one of the following value methods to retrieve string values:

To obtain all string values	`getStringValues()`
To obtain the string values of a specified child property	`getChildPropValue()`

- If `hasValue()` returns `false`, use one of the following value methods to retrieve any child properties:

To obtain all child properties	`getHierChildProps()`
To obtain a specified child property	`getHierChildProp()`
To obtain all child properties whose names begin with a specified prefice	`getChildPropsWithPrefix()`

See also

hasChildren(),

setEncryptionFlag()

Sets the encryption flag of a connector configuration property in its connector-property object.

Syntax

```
public void setEncryptionFlag(boolean encryptFlag);
```

Parameters

encryptFlag Is a boolean value to indicate whether the current connector configuration property's value should be encrypted.

Return values

None.

Exceptions

None.

Notes

The `setEncryptionFlag()` method sets the boolean encryption flag from the connector-property object. This flag indicates whether the connector property's string values are encrypted.

Note: In Connector Configurator, encrypted values display as a string of asterisk (*) characters.

See also

getEncryptionFlag()

setValues()

Sets the values of the hierarchical connector configuration property.

Syntax

```
public void setValues(Object[] propValues);
```

Parameters

propValues Is an array of `Object` values, each array element is a single property value.

Return values

None.

Exceptions

None.

Notes

The `setValues()` method allows you to set the values of a hierarchical connector configuration property. You specify the property values in the *`propValues`* array, which is an array of `Objects`. Therefore, you can pass both string and child-property values in this single array. Make sure you assign property values in the *`propValues`* array in the order that they are defined within the hierarchical connector property.

For example, the following call to `setValues()` assigns both a string value and a child property to the connector property in `topLevelProp`:

```
Object[] propValues;
CWProperty childProp;

propValues[0] = "stringValue"
propValue[1] = childProp;
topLevelProp.setValues(propValues);
```

See also

getHierChildProp(),, getHierChildProps(),, getStringValues()

Part 4. Java low-level connector library API reference

Chapter 26. Overview of the low-level Java connector library

The low-level Java connector library includes the low-level class libraries on which the high-level Java connector library is based. This connector class library contains predefined classes for low-level Java connectors. The low-level Java connector library also provide utilities, such as methods to implement tracing and logging services.

Important: The low-level Java connector library is a deprecated library for the development of Java connectors. For development of new Java connectors, use the Java connector library. For more information on the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

IBM includes the predefined classes and interfaces of the low-level Java connector library in the product Java jar (Java archive file), `crossworlds.jar`. The `crossworlds.jar` file resides in the `bin` subdirectory of the product directory.

Classes and interfaces

Table 145 lists the classes and interfaces in the low-level Java connector library.

Table 145. Classes and interfaces in the low-level Java connector library

Class or interface	Description	Page
`BOHandlerBase`	Represents the base class for a business object handler. You extend this class to define one or more business object handler for your connector.	407
`BusinessObjectInterface`	Represents a business object instance. It provides access to the names and values of attributes	411
`ConnectorBase`	Represents the base class for a connector. You extend this class to define your connector class and implement the required virtual methods	427
`CxObjectContainerInterface`	Manages an array of child business objects	445
`CxObjectAttr`	Represents an attribute descriptor, which contains information about the properties of an attribute	437
`CxProperty`	Represents a connector-property object, which contains a hierarchical connector configuration property	449
`CxStatusConstants`	Defines outcome-status constants for use with the low-level Java connector library	457
`JavaConnectorUtil`	Provides miscellaneous utility methods for use in a Java connector; These utility methods fall into the following general categories: • Static methods for generating and logging messages • Static methods for creating business objects • Static methods for obtaining connector configuration properties • Methods for obtaining locale information	459
`ReturnStatusDescriptor`	Represents a return-status descriptor, which contains error and informational messages	473
Exceptions	Exception subclasses represent exceptions that methods of the low-level Java connector library throw	475

Chapter 27. BOHandlerBase class

The BOHandlerBase class is the low-level Java connector library class for the base class of a business object handler. It is part of the AppSide_Connector package. All low-level Java connectors must extend this class for each of its business object handlers and implement the doVerbFor() method in each of the derived classes.

Note: The CWConnectorBOHandler class is the Java connector library method that is a wrapper for the BOHandlerBase class of the low-level Java connector library. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

The connector framework calls ConnectorBase.getBOHandlerForBO() to create a business object handler for each of the business object definitions that the connector supports.

Table 146 summarizes the methods in the BOHandlerBase class.

Table 146. Member methods of the BOHandlerBase class

Member method	Description	Page
doVerbFor()	Performs the action for the active verb of a business object.	407
getName()	Returns the name of the business object handler.	408
setName()	Sets the name of the business object handler.	409

doVerbFor()

Performs the action for the active verb of a business object. This method is the primary public interface for the business object handler.

Syntax

```
public int doVerbFor(BusinessObjectInterface theBusObj,
    ReturnStatusDescriptor rtnObj);
```

Parameters

theBusObj Is the incoming business object.

rtnObj Is the status descriptor object that contains an error or informational message for the integration broker and the status of the operation.

Return values

An integer that indicates the outcome status of the verb operation:

CxStatusConstants.SUCCEED
: The verb operation succeeded.

CxStatusConstants.FAIL
: The verb operation failed.

`CxStatusConstants.APPRESPONSETIMEOUT`
The application is not responding.

`CxStatusConstants.VALCHANGE`
At least one value in the business object changed.

`CxStatusConstants.VALDUPES`
The requested operation found multiple records for the same key value.

`CxStatusConstants.MULTIPLE_HITS`
The connector finds multiple matching records when retrieving using non-key values. The connector will only return the first matching record in a business object.

`CxStatusConstants.RETRIEVEBYCONTENT_FAILED`
The connector was not able to find matches for Retrieve by non-key values.

`CxStatusConstants.BO_DOES_NOT_EXIST`
The requested business object entity does not exist in the database.

Notes

When a business object arrives from the integration broker, the connector framework creates a status descriptor object and sends it as an argument in its call to the `doVerbFor()` method, which performs the action of the business object's active verb.

Important: The `doVerbFor()` method is an abstract method that you *must* implement for the business object handler.

The `doVerbFor()` method should take the following steps:

- Perform the verb operation.
- Call `ReturnStatusDescriptor.setErrorString()` to set a message in the status descriptor object if there is an informational, warning, or error return message.
- Call `ReturnStatusDescriptor.setStatus()` to return a status return code. The `setStatus()` method takes an integer whose value should be the same as the return value of the `doVerbFor()` method.

See also

See also the description of the BusinessObjectInterface class.

---

getName()

Retrieves the name of the `BOHandler` object.

Syntax

```
protected String getName();
```

Parameters

None.

Return values

A `String` containing the name of the `BOHandler` object. If `setName()` has not been called on the `BOHandlerBase` instance prior to this method, returns null.

See also

See also the `setName()` method.

---

setName()

Sets the name of the `BOHandler` object, the business object handler. This name is typically the name of the business object the handler has been created to process.

Syntax

```
protected void setName(String name);
```

Parameters

name Specifies the name of the `BOHandler` object.

Return values

None.

Chapter 28. BusinessObjectInterface interface

The BusinessObjectInterface interface gives a view of the business object to the developers of low-level Java connectors. It is part of the CxCommon package. The interface defines methods for getting information about the metadata of the business object, and methods for reading and modifying the business object instance. Each instance of BusinessObjectInterface represents a single business object.

Note: The CWConnectorBusObj class is the Java connector library class that provides the functionality for the BusinessObjectInterface interface of the low-level Java connector library. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

Important: The low-level Java connector library provides an implementation of this interface internally. *Connector developers should not implement this class.*

Table 147 summarizes the methods in the BusinessObjectInterface interface.

Table 147. Member methods of the BusinessObjectInterface interface

Member method	Description	Page
clone()	Copies an existing business object.	412
doVerbFor()	Calls the business object handler (instance of the BOHandlerBase class) to perform the actions of the business object's active verb.	412
dump()	Formats and returns the business object information in a standard defined format for logging and tracing.	413
getAppText()	Retrieves the value of the business object's AppSpecificInfo field	414
getAttrCount()	Retrieves the number of attributes that the business object has.	414
getAttrDesc()	Retrieves an attribute description by name or by position.	414
getAttribute()	Retrieves the attribute value.	415
getAttributeIndex()	Retrieves the index position of a given attribute.	415
getAttributeType()	Retrieves the attribute type code for a given attribute using the attribute name or the attribute's position.	416
getAttrName()	Retrieves the name of an attribute by position.	416
getAttrValue()	Retrieves an attribute value by name or by position.	417
getBusinessObjectVersion()	Retrieves the version of the business object.	417
getDefaultAttrValue()	Retrieves the default value of an attribute value by name or by position.	418
getLocale()	Retrieves the locale associated with the business object.	418
getName()	Retrieves the name of the business object specification that the business object references.	419
getParentBusinessObject()	Retrieves the parent business object of the current business object.	419
getVerb()	Retrieves the active verb for the business object.	420
getVerbAppText()	Retrieves the verb application-specific information.	420

Table 147. Member methods of the BusinessObjectInterface interface (continued)

Member method	Description	Page
isBlank()	Determines whether the value of the attribute with the specified name or position is blank.	420
isIgnore()	Determines whether the value of the attribute with the specified name or position is "ignore".	421
isVerbSupported()	Determines whether a verb is supported or not.	421
makeNewAttrObject()	Creates a new object of the correct type for the attribute with the specified name or position. This operation applies typically to attributes that contain child objects.	421
setAttributeWithCreate()	Sets an object's attribute value.	422
setAttrValue()	Sets the value of an attribute by name or by position.	423
setDefaultAttrValues()	Initializes the business object's attributes with their default values.	424
setLocale()	Sets the locale associated with the business object.	424
setVerb()	Sets the active verb for the business object.	425

clone()

Copies an existing business object. It copies the business object attributes and also its verb.

Syntax

```
public Object clone();
```

Parameters

None.

Return values

A copy of the current business object, including its attributes and verbs.

doVerbFor()

Invokes the business object handler to perform the action specified by the active verb in the business object.

Syntax

```
public int doVerbFor(ReturnStatusDescriptor rtnObj);
```

Parameters

rtnObj Is the status descriptor object that contains an error or informational message for the execution of this method. The integration broker uses this message.

Return values

An integer that specifies the outcome status of the verb operation.

`CxStatusConstants.SUCCEED`
The verb operation succeeded.

`CxStatusConstants.FAIL`
The verb operation failed.

`CxStatusConstants.APPRESPONSETIMEOUT`
The application is not responding.

`CxStatusConstants.VALCHANGE`
At least one value in the business object changed.

`CxStatusConstants.VALDUPES`
The requested operation found multiple records for the same key value.

`CxStatusConstants.MULTIPLE_HITS`
The connector finds multiple matching records when retrieving with non-key values. The connector will only return the first matching record in a business object.

`CxStatusConstants.RETRIEVEBYCONTENT_FAILED`
The connector was not able to find matches for Retrieve by non-key values.

`CxStatusConstants.BO_DOES_NOT_EXIST`
The requested business object entity does not exist in the database.

Notes

The execution of this method sets the passed-in parameter with the error or informational message. The message is then sent back to the integration broker.

The business object provides all the operations for the verbs that the business object definition supports.

The active verb is one of the list of verbs that the business object definition contains. To determine the active verb for a business object, you can use the `getVerb()` method.

See also

See also the descriptions of the `getVerb()` and `setVerb()` methods and the `BusinessObjectInterface` interface.

dump()

Returns business object information in a readable format for logging and tracing.

Syntax

```
public String dump();
```

Parameters

None.

Return values

A `String` that contains the formatted business object information.

getAppText()

Retrieves the application-specific information for this business object definition.

Syntax

```
public String getAppText();
```

Parameters

None.

Return values

A `String` object that holds the value of the `AppSpecificInfo` field for the business object. This method can return `null`.

getAttrCount()

Retrieves the number of attributes that are in the business object's attribute list.

Syntax

```
public int getAttrCount();
```

Parameters

None.

Return values

An integer that specifies the number of attributes in the attribute list.

See also

See also the description of the `getAttrIndex()` method.

getAttrDesc()

Retrieves the description of an attribute of a business object, given the attribute's name or position.

Syntax

```
public CxObjectAttr getAttrDesc(String name);

public CxObjectAttr getAttrDesc(int position);
```

Parameters

name	Is the name of the attribute in the business object definition.
position	Is the position of the attribute in the business object definition.

Return values

A `CxObjectAttr` object that defines the specified attribute.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the name or position specified is not valid for the definition of this business object.

Notes

To retrieve the description of an attribute of the business object, specify either the attribute name or its position in the list of attributes:

- The first form of the `getAttrDesc()` method retrieves the description of an attribute of a business object, given the attribute's name.
- The second form retrieves the description of an attribute of a business object, given its position within the business object definition.

See also

See also the description of the `getAttrName()` method.

---

getAttribute()

Retrieves the value of an attribute, given the attribute's name.

Syntax

```
public Object getAttribute(String attrName);
```

Parameters

attrName Is the name of the attribute in the business object definition.

Return values

An `Object` that contains the attribute value.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the name specified is not valid for the definition of this business object.

Notes

This method differs from `getAttrValue()` in that `getAttribute()` can do a deep retrieve of attribute values. For example, if a Customer business object contains an Address business object, `getAttribute()` can retrieve an AddressId from the Address subobject, at the fifth position in the container: Address[4].AddressId.

---

getAttributeIndex()

Retrieves the ordinal position of a given attribute of a business object.

Syntax

```
public int getAttributeIndex(String name);
```

Parameters

name Is the name of the attribute in the business object definition.

Return values

The integer ordinal position of the attribute.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the name specified is not valid for the definition of this business object.

getAttributeType()

Retrieves the data type of the attribute using either the ordinal position of a given attribute of a business object or the name of the attribute.

Syntax

```
public int getAttributeType(String name);
public int getAttributeType(int position);
```

Parameters

name Is the name of the attribute in the business object definition.

position Is the ordinal position of the attribute in the business object definition.

Return values

The type of the attribute, represented as an integer. See Table 151 on page 437 for the possible attribute-type constants.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the name or position specified is not valid for the definition of this business object.

Notes

To retrieve the type of an attribute of the business object, specify either the attribute name or its position in the list of attributes.

getAttrName()

Retrieves the name of an attribute that you specify by its position in the business object's attribute list.

Syntax

```
public String getAttrName(int position);
```

Parameters

position Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

Return values

A `String` that contains the name of the specified attribute.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the position specified is not valid for the definition of this business object.

getAttrValue()

Retrieves the value of an attribute of a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public Object getAttrValue(String name);
public Object getAttrValue(int position);
```

Parameters

name Is the name of an attribute.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

An `Object` that contains the value of the specified attribute, in the format defined for the attribute's data type.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the position or name specified is not valid for the definition of this business object.

Notes

The `getAttrValue()` method returns a `java.lang.Object`, which you cast to the proper type before assigning to a variable.

See also

See also the description of the `getAttrName()` method.

getBusinessObjectVersion()

Retrieves the version of the business object definition. The version is represented by the major, minor, and point components -x.y.z. For example: - 1.0.2.

Syntax

```
public String getBusinessObjectVersion();
```

Parameters

None.

Return values

A `String` that contains the version number of the business object.

getDefaultAttrValue()

Retrieves the default value of an attribute of a business object, given the attribute's name or its position in the business object's attribute list.

Syntax

```
public String getDefaultAttrValue(int position);
public String getDefaultAttrValue(String name);
```

Parameters

name Is the name of an attribute.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The `String` containing the default value of the attribute. If no default value exists for the attribute, the method returns an empty string.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the position or name specified is not valid for the definition of this business object.

Notes

To retrieve the default value of an attribute of the business object, specify either the attribute name or the attribute's position in the list of attributes.

See also

See also the description of the `getAttrValue()` method.

getLocale()

Retrieves the locale associated with the business object.

Syntax

```
public Locale getLocale();
```

Parameters

None.

Return values

A Java `Locale` object that describes the locale associated with the current business object.

Notes

The `getLocale()` method returns the business-object locale, which is associated with the business object. This locale indicates the language and code encoding associated with the data in the business object, *not* with the name of the business object definition or its attributes (which must be characters in the code set associated with the U.S. English locale, `en_US`). If the business object does not have a locale associated with it, the connector framework assigns the connector-framework locale as the business-object locale.

See also

createBusObj(), getGlobalLocale(), setLocale()

getName()

Retrieves the name of the business object definition that the business object references.

Syntax

```
public String getName();
```

Parameters

None.

Return values

A `String` object containing the name of a business object definition.

See also

See also the description of the `getBusinessObjectVersion()` method.

getParentBusinessObject()

Retrieves the parent business object of the current business object. If this business object instance is a root object, in which case it has no parent object, then the method returns `null`.

Syntax

```
public BusinessObjectInterface getParentBusinessObject();
```

Parameters

None.

Return values

The business object that contains the parent business object, or `null` if the current business object is a root and has no parent.

getVerb()

Retrieves the active verb for the business object.

Syntax

```
public String getVerb();
```

Parameters

None.

Return values

A String object that contains the active verb for the business object. If there is no active verb for the business object, the returned `String` is empty.

Notes

The business object definition contains the list of verbs that the business object supports. The `getVerb()` method enables you to determine which verb is active for the business object.

See also

See also the description of the `setVerb()` method.

getVerbAppText()

Retrieves the value of the application-specific information for a particular verb.

Syntax

```
public String getVerbAppText(String verb);
```

Parameters

verb Is the verb for which the value of the `AppSpecificInfo` field is to be retrieved.

Return values

A `String` object containing the value of `AppSpecificInfo` for the specified verb. If the business object does not have application-specific information for the verb, the method returns an empty string.

See also

See also the description of the `getVerb()` method.

isBlank()

Determines whether the value is blank for the attribute with the specified name or at the specified position in the attribute list.

Syntax

```
public boolean isBlank(int position);
public boolean isBlank(String name);
```

Parameters

name Is the name of an attribute.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns True if the attribute value equals the blank value or False if it does not.

isIgnore()

Determines whether the value is Ignore for the attribute with the specified name or at the specified position in the attribute list.

Syntax

```
public boolean isIgnore(int position);
public boolean isIgnore(String name);
```

Parameters

name Is the name of an attribute.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

Returns `true` if the attribute value equals the special Ignore value or `false` if it does not.

isVerbSupported()

Determines whether or not the verb passed to the method is supported by this business object definition.

Syntax

```
public boolean isVerbSupported(String verb);
```

Parameters

verb Is the verb which the method determines if supported.

Return values

Returns `true` if the passed-in verb is supported; otherwise, returns `false`.

See also

See also the descriptions of the `getVerb()` method.

makeNewAttrObject()

Creates a new business object of the correct type for the attribute.

Syntax

```
public Object makeNewAttrObject(int position);
public Object makeNewAttrObject(String name);
```

Parameters

name Is the name of an attribute.

position Is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

An `Object` containing the newly created instance of the attribute class.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the position or name specified is not valid for the definition of this business object.

Notes

The `makeNewAttrObject()` method creates a new business object of the correct type for the attribute with the specified name or at the specified position in the attribute list. For example, for an attribute of type container, the method returns an instance of a `CxObjectContainerInterface`.

The caller needs to cast the returned object to the correct type. In the case where the type is a business object, the caller must cast the returned object to BusinessObjectInterface. For an attribute whose value is a container, cast the returned object to `CxObjectContainerInterface`.

This method should typically be used with attributes that contain child objects.

setAttributeWithCreate()

Sets an object's attribute, creating the object's attributes regardless of the intervening objects and containers.

Syntax

```
public void setAttributeWithCreate(String attrName, Object value);
```

Parameters

attrName Is the name of the attribute to set.

value Is the attribute value.

Return values

None.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the position or name specified is not valid for the definition of this business object.

`CxObjectInvalidAttrException`
Thrown if the value passed in is not a valid value for the particular attribute.

`BusObjSpecNameNotFoundException`
Thrown if the business object definition is not found in the database.

Notes

The `setAttributeWithCreate()` method forcibly sets an object's attribute; that is, it creates the object's attributes regardless of the intervening objects and containers. The supported grammar is: `attr1.attr2...attrThatsAContainer[index]...attrN`. For example, `Address[5].AddressObjId` refers to the object identifier of the fifth element in the business object array referenced by the Address attribute.

---

setAttrValue()

Sets the value of an attribute.

Syntax

```
public void setAttrValue(String attrName, Object newval);

public void setAttrValue(int position, Object newval);
```

Parameters

attrName Is the name of the attribute whose value you want to set.

position Is an integer that specifies the ordinal position of the attribute in the business object's attribute list.

newval Is the value to set in the business object.

Return values

None.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the position or name specified is not valid for the definition of this business object.

`CxObjectInvalidAttrException`
Thrown if the value passed in is not a valid value for the particular attribute.

Notes

You can use the name or position method to set an attribute value.

The `setAttrValue()` method sets the value of an attribute to the value passed in as a parameter to the method. This value can be of any type supported by the IBM WebSphere business integration system. If the attribute type is a type other than container and subobject type, then the passed-in parameter is of type String. For subobjects, the passed-in parameter is of type BusinessObjectInterface. For containers, the passed-in parameter can be either of type CxObjectContainerInterface or BusinessObjectInterface.

This method can be called directly on a container attribute with an instance of type `BusinessObjectInterface`. When this is the first business object that this container holds, a container will be created internally and this business object is inserted into that new container. Subsequent similar calls add business objects to the same container. Alternatively, you can create a container of type `CxObjectContainerInterface` by using the `JavaConnectorUtil.createContainer()` method, then inserting all business objects into this container and invoking `setAttrValue()` with the container as the parameter.

See also

See also the description of the `getDefaultAttrValues()` method.

setDefaultAttrValues()

Sets default values for attributes which currently have the special Blank or Ignore attribute values.

Syntax

```
public void setDefaultAttrValues();
```

Parameters

None.

Return values

None.

Notes

The default values are valid values, not ignore values. For attributes whose type is container, the method creates an empty container. The method sets default values for instances of sub-objects within the business object.

See also

See also the description of the `setAttrValue()` method.

setLocale()

Sets the locale for the business object.

Syntax

```
public void setLocale(Locale localeObj);
public void setLocale(String localeName);
```

Parameters

localeName Is the name of the locale to associate with the current business object.

localeObj Is a Java `Locale` object that describes the locale to associate with the current business object.

Return values

None.

Exceptions

`IllegalLocaleException`
Thrown if the locale name specified is not valid.

Notes

The `setLocale()` method sets the business-object locale, which identifies the locale that is associated with the business object. This locale indicates the language and code encoding associated with the data in the business object, *not* with the name of the business object definition or its attributes (which must be characters in the code set associated with the U.S. English locale, `en_US`). If the business object does not have a locale associated with it, the connector framework assigns the connector-framework locale as the business-object locale.

See also

getLocale()

---

setVerb()

Sets the active verb for a business object.

Syntax

```
public void setVerb(String newVerb);
```

Parameters

newVerb Is a verb that is in the verb list of the business object definition to which the business object refers.

Return values

None.

Exceptions

`BusObjInvalidVerbException`
Thrown if the verb passed in is not a valid verb in the business object definition.

Notes

The business object definition contains the list of verbs that the business object supports. The verb that you set as the active verb must be on this list. Only one verb is active at a time for a business object.

All business objects typically support the Create, Retrieve, and Update verbs. A business object might support additional verbs, such as Delete. Every connector that supports the business object must implement all the verbs that it supports.

See also

See also the descriptions of the `getVerb()` method.

Chapter 29. ConnectorBase class

The ConnectorBase class is the base class for a low-level Java connector. It is part of the AppSide_Connector package. From this class, a connector developer must derive a connector class and implement the abstract methods for the connector. This derived class contains the code for the application-specific component of the connector.

Note: The CWConnectorAgent class is the Java connector library method that is a wrapper for the ConnectorBase class of the low-level Java connector library. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

Important: All low-level Java connectors *must* extend this abstract class, which contains the following abstract methods: init(), getVersion(), getBOHandlerForBO(), pollForEvents(), and terminate(). Developers *must* provide implementations for these abstract methods.

Table 148 summarizes the methods in the ConnectorBase class.

Table 148. Member methods of the ConnectorBase class

Member method	Description	Page
executeCollaboration()	Sends business object request to a collaboration.	427
getBOHandlerForBO()	Retrieves the handler for a business object.	428
getCollabNames()	Retrieves a list of collaboration names that are available to process business object requests.	428
getSupportedBusObjNames()	Retrieves a list of supported business objects for the connector.	429
getVersion()	Retrieves the version of the application connector.	429
gotApplEvent()	Sends a business object to InterChange Server.	430
init()	Initializes the connector and establishes a connection with the application.	431
isAgentCapableOfPolling()	Determines whether this connector-agent process can perform polling.	432
isSubscribed()	Checks if subscriptions exist for the business object and verb combination.	433
pollForEvents()	Polls an application for changes to business objects.	434
terminate()	Closes the connection with the application and frees allocated resources.	435

executeCollaboration()

Sends business object requests to collaborations. This is a synchronous request.

WebSphere InterChange Server
This method is only valid when the integration broker is InterChange Server.

Syntax

```
public void executeCollaboration(String collabName,
   BusinessObjectInterface theBusObj, ReturnStatusDescriptor rtnStatus);
```

Parameters

collabName	Specifies the name of the collaboration that should execute the business object request.
theBusObj	Is the incoming and returned business object.
rtnStatus	Is the status descriptor containing a message and the execution or return status from the collaboration.

Return values

None.

See also

See also the description of the `BusinessObjectInterface`.

getBOHandlerForBO()

Retrieves the business object handler for a business object definition.

Syntax

```
public BOHandlerBase getBOHandlerForBO(String busObjName);
```

Parameters

busObjName	Is the name of a business object.

Return values

A reference to a business object handler.

Notes

The connector framework calls the `getBOHandlerForBO()` method to retrieve the business object handler for a business object definition.

Important: The `getBOHandlerForBO()` method is an abstract method that you *must* implement for the connector.

You can use one business object handler for multiple business object definitions or a business object handler for each business object definition.

getCollabNames()

Retrieves the list of collaborations that are available to process business object requests.

WebSphere InterChange Server

This method is only valid when the integration broker is InterChange Server.

Syntax

```
public String [] getCollabNames();
```

Parameters

None.

Return values

An array of `String` objects containing a list of collaboration names.

getSupportedBusObjNames()

Retrieves a list of supported business objects for the current connector.

Syntax

```
public String[] getSupportedBusObjNames()
```

Parameters

None.

Return values

A `String` array that contains a list of the names of the supported business objects for the connector.

Notes

The `getSupportedBusObjNames()` method returns a list of top-level supported business objects for the current connector; that is, if the connector supports business objects that contain child business objects, `getSupportedBusObjNames()` includes only the name of the parent object in its list.

getVersion()

Retrieves the version of the connector.

Syntax

```
public String getVersion();
```

Parameters

None.

Return values

A `String` indicating the version of the connector's application-specific component.

Notes

The connector framework calls the `getVersion()` method to retrieve the version of the connector.

Important: The `getVersion()` method is an abstract method that you *must* implement for the connector.

gotApplEvent()

Sends a business object to the connector framework.

Syntax

```
public int gotApplEvent(BusinessObjectInterface theBusObj);
```

Parameters

theBusObj Is the business object instance being sent to the connector framework.

Return values

An integer that indicates the outcome status of the event delivery. Compare this integer value with the following outcome-status constants to determine the status:

`CxStatusConstants.SUCCEED`
The connector framework successfully delivered the business object to the connector framework.

`CxStatusConstants.FAIL`
The event delivery failed.

`CxStatusConstants.CONNECTOR_NOT_ACTIVE`
The connector is paused and therefore unable to receive events.

`CxStatusConstants.NO_SUBSCRIPTION_FOUND`
No subscriptions exist for the event that the business object represents.

Notes

The `gotApplEvent()` method sends the *theBusObj* business object to the connector framework. The connector framework does some processing on the event object to serialize the data and ensure that it is persisted properly. It then makes sure the event is either sent to the ICS through IIOP or written to a queue (if you are using queues for event notification).

WebSphere InterChange Server
If the integration broker is InterChange Server, the connector framework sends the event (as a business object) to InterChange Server across its configured delivery transport mechanism (such as JMS or CORBA IIOP).

Other integration brokers
If the integration broker is a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework sends the event (as an XML message) to WebSphere MQ Integrator Broker across its configured delivery transport mechanism of a JMS queue.

Before sending the business object to the connector framework, `gotApplEvent()` checks for the following conditions and returns the associated outcome status if

these conditions are *not* met:

Condition	Outcome status
Is the status of the connector active; that is, it is not in a "paused" state? When the connector's application-specific component is paused, it no longer polls the application.	CONNECTOR_NOT_ACTIVE
Is there a subscription for the event?	NO_SUBSCRIPTION_FOUND

Note: Because `gotApplEvent()` makes sure that the business object and verb to be sent have a valid subscription, you do *not* need to call `isSubscribed()` immediately before calling `gotApplEvent()`.

WebSphere InterChange Server

Usually, you call the `gotApplEvent()` method from the `pollForEvents()` thread. InterChange Server uses the `pollForEvents()` method to request the connector to send subscribed events to it. The connector uses the `gotApplEvent()` method to send business objects to the connector framework, which in turn routes them to InterChange Server in response.

The connector uses the `pollForEvents()` method to poll the event store for subscribed events to send to the integration broker. Within `pollForEvents()`, the connector uses the `gotApplEvent()` method to send an event (represented as a business object) to the connector framework. The connector framework then routes this business object to the integration broker. Therefore, the poll method should check the return code from `gotApplEvent()` to ensure that any errors that are returned are handled appropriately. For example, until the event delivery is successful, the poll method should *not* remove the event from the event store. Instead, the poll method should update the event record's status to reflect the results of the event delivery based on the return code of `gotApplEvent()`.

The `gotApplEvent()` method initiates an asynchronous execution of an event. Asynchronous execution means that the calling code does *not* wait for receipt of the event, nor does it wait for a response.

Note: To initiate a sychronous execution of an event, use the `executeCollaboration()` method. Sychronous execution means that the calling code waits for the receipt of the event, and for a response.

See also

executeCollaboration(), isSubscribed(), pollForEvents()

See also the description of the `BusinessObjectInterface` interface.

init()

Initializes the connector.

Syntax

```
public int init();
```

Parameters

None.

Return values

An integer that indicates the status of the initialization operation. If the initialization operation succeeds, returns `CxStatusConstants.SUCCEED`; otherwise, it returns a negative value. Possible failure values are:

`CxStatusConstants.FAIL`
: Initialization failed.

`CxStatusConstants.UNABLETOLOGIN`
: The connector is unable to log in to the application.

Notes

The connector framework calls the `init()` method to initialize the connector when the connector starts up. Be sure to implement all of the initialization for the connector, such as logging on to an application, in the `init()` method.

Important: The `init()` method is an abstract method that you *must* implement for the connector.

isAgentCapableOfPolling()

Determines whether a connector process is capable of polling.

WebSphere InterChange Server
This method is only valid when the integration broker is InterChange Server.

Syntax

```
boolean isAgentCapableOfPolling();
```

Parameters

None.

Return values

A `boolean` value that indicates whether the connector is capable of polling. This return value depends on the type of connector:

Connector process type	Return value
Master (serial processing)	`true`
Master (parallel processing)	`false`
Slave (request)	`false`
Slave (polling)	`true`

Notes

If a connector is configured to run in the single-process mode (with `ParallelProcessDegree` set to 1, which is the default), the `isAgentCapableOfPolling()` method always returns `true` because the same connector process performs both event polling and request processing.

If a connector is configured to run in parallel-process mode (with `ParallelProcessDegree` greater than 1), it consists of several processes, each with a particular purpose, as shown in Table 149.

Table 149. Purposes of processes of a parallel connector

Connector process	Purpose of connector process
Connector-agent master process	Receives the incoming event from ICS and determines to which of the connector's slave processes to route the event
Request-processing slave process	Handles requests for the connector
Polling slave process	Handles polling and event delivery for the connector

The return value of `isAgentCapableOfPolling()` depends on the purpose of the connector-agent process that makes the call to this method. For a parallel-process connector, this method returns `true` *only* when called from a connector whose purpose is to serve as a polling slave. For more information on parallel-process connectors, see the *System Administration Guide*.

isSubscribed()

Determines whether the integration broker has subscribed to a particular business object with a particular verb.

Syntax

```
public boolean isSubscribed(String busObjName, String verb);
```

Parameters

busObjName Is the name of a business object.

verb is the active verb for the business object.

Return values

Returns `true` if the integration broker is interested in receiving the specified business object and verb; otherwise, returns `false`.

Notes

The `isSubscribed()` method is part of the subscription manager, which tracks all subscribe and unsubscribe messages that arrive from the connector framework and maintains a list of active business object subscriptions. For a Java connector, this subscription manager is part of the connector base class.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the poll method can determine if any collaboration subscribes to the *busObjName* business object with the specified *verb*. At initialization, the connector framework requests its subscription list from the connector controller. At runtime, the poll method can use `isSubscribed()` to query the connector framework to verify that some collaboration subscribes to a particular business object. The poll method can send the event only if some collaboration is currently subscribed. For more information, see "Business object subscription and publishing" on page 13.

Other integration brokers

If your business integration system uses a WebSphere message broker (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, the connector framework assumes that the integration broker is interested in *all* the connector's supported business objects. If the application-specific component uses the `isSubscribed()` method to query the connector framework about subscriptions for a particular business object, the method returns `true` for *every* business object that the connector supports.

See also

gotApplEvent(), pollForEvents()

---

pollForEvents()

Polls an application's event store for events that cause changes to business objects.

Syntax

```
public int pollForEvents();
```

Parameters

None.

Return values

An integer that indicates the outcome status of the polling operation.The following return codes are typically used by the `pollForEvents()` method.

`CxStatusConstants.SUCCEED`
: The polling action succeeded regardless of whether an event is retrieved.

`CxStatusConstants.FAIL`
: The polling operation failed.

`CxStatusConstants.APPRESPONSETIMEOUT`
: The application is not responding.

Notes

The connector infrastructure calls the `pollForEvents()` method, at a time interval that you can configure, so that the connector can detect any event in the application that is interesting to a subscriber. The frequency at which the class library calls this method depends on the poll frequency value that is configured by the `PollFrequency` connector configuration property.

Important: The `pollForEvents()` method is an abstract method that you *must* implement to provide your own polling mechanism.

Note: If your connector executes in a parallel-process mode, it uses a separate polling slave process to handle polling.

terminate()

Performs clean-up operations when the connector is shutting down.

Syntax

```
public int terminate();
```

Parameters

None.

Return values

An integer that indicates the status value of the `terminate()` operation.

`CxStatusConstants.SUCCEED`
: The terminate operation succeeded.

`CxStatusConstants.FAIL`
: The terminate operation failed.

Notes

The connector infrastructure calls the `terminate()` method when the connector is shutting down. In your implementation of this method, it is good practice to free all the memory and log off from the application.

Important: The `terminate()` method is an abstract method that you *must* implement for the connector.

Deprecated methods

Some methods in the `ConnectorBase` class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but IBM recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 150 lists the deprecated methods for the `ConnectorBase` class. If you are writing a new connector (not modifying an existing connector), you can ignore this section.

Table 150. Deprecated methods of the ConnectorBase class

Former method	Replacement
`consumeSync()`	`executeCollaboration()`

Chapter 30. CxObjectAttr class

The CxObjectAttr class is the object attribute class for Java connectors. It is part of the CxCommon package. It defines an attribute of a business object specification. The class defines methods for getting information about attributes.

Note: The CWConnectorBusObj class is the Java connector library class that is a wrapper for the methods in the CxObjectAttr class of the low-level Java connector library. The CWConnectorBusObj class provides access to a business object, business object array, business object definition, and attributes. The CWConnectorAttrType class defines the attribute-type constants. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

This class contains the following:

- "Attribute-type constants"
- "Methods"

Attribute-type constants

The CxObjectAttr class defines numeric and string equivalents for attribute types, as shown in Table 151.

Table 151. Numeric and string equivalents for attribute types

Attribute type	Equivalent numeric constants	Equivalent string constants
Boolean	BOOLEAN	BOOLSTRING
Business object: multiple cardinality		MULTIPLECARDSTRING
Business object: single cardinality		SINGLECARDSTRING
Date	DATE	DATESTRING
Double	DOUBLE	DOUBSTRING
Float	FLOAT	FLTSTRING
Integer	INTEGER	INTSTRING
Long text	LONGTEXT	LONGTEXTSTRING
Object	OBJECT	
String	STRING	STRSTRING
Invalid type number	INVALID_TYPE_NUM	INVALID_TYPE_STRING

Methods

Table 152 summarizes the methods in the CxObjectAttr interface.

Table 152. Member methods of the CxObjectAttr class

Member method	Description	Page
equals()	Determines if a specified attribute is the same as this attribute	438
getAppText()	Retrieves the application specific information of this attribute.	438
getCardinality()	Retrieves the cardinality of an attribute.	439
getDefault()	Retrieves an attribute's default value.	439

Table 152. Member methods of the CxObjectAttr class (continued)

Member method	Description	Page
getMaxLength()	Retrieves the maximum length of an attribute value.	439
getName()	Retrieves the attribute name.	440
getRelationType()	Retrieves the type of an attribute relationship.	440
getTypeName()	Retrieves the type of an attribute.	440
getTypeNum()	Retrieves the numeric type code of an attribute.	440
hasCardinality()	Compares the cardinality of an attribute with the cardinality value passed in as a parameter.	441
hasName()	Compares the name passed in to the method to the attribute's name.	441
hasType()	Verifies if the type of the attribute is the same as the type passed in.	441
isForeignKeyAttr()	Verifies if the attribute is part of the object's foreign key.	442
isKeyAttr()	Verifies if the attribute is part of the object's key set.	442
isMultipleCard()	Retrieves whether the attribute is a multiple cardinality.	442
isObjectType()	Retrieves if the attribute type is an object type.	443
isRequiredAttr()	Verifies if this attribute is a required attribute for the business object.	443
isType()	Verifies if the attribute type matches the passed-in parameter value.	443

equals()

Determines whether a specified attribute is the same as the current attribute.

Syntax

```
public boolean equals(Object obj)
```

Parameters

obj Is the object that represents the attribute to compare with the current attribute.

Return values

Returns True if the specified attribute is the same as this attribute; otherwise returns False.

Notes

This method verifies if the specified attribute matches in name, type, whether it is a key, whether it is a foreign key and whether it is a required attribute matches, with this attribute.

getAppText()

Retrieves the application-specific information of this attribute.

Syntax

```
public String getAppText();
```

Parameters

None.

Return values

A `String` object that holds the value of the `AppSpecificText` field for the attribute. If the attribute does not have any application-specific information, this method returns `null`.

getCardinality()

Retrieves the cardinality of an attribute.

Syntax

```
public String getCardinality();
```

Parameters

None.

Return values

A `String` containing the cardinality of the attribute. The value of the string is either 1 or n.

getDefault()

Retrieves the default value for this attribute.

Syntax

```
public String getDefault();
```

Parameters

None.

Return values

A `String` containing the default value of the attribute, or `null`.

getMaxLength()

Retrieves the maximum length of an attribute from the business object definition.

Syntax

```
int getMaxLength();
```

Parameters

None.

Return values

An integer that specifies the maximum length, in bytes, that an attribute value can have.

getName()

Retrieves the name of the attribute.

Syntax

```
public String getName();
```

Parameters

None.

Return values

A `String` containing the name of the specified attribute.

getRelationType()

Retrieves the relationship type of an attribute. For complex attributes (such as subobjects and arrays) the returned relationship type is a containment relationship.

Syntax

```
public String getRelationType();
```

Parameters

None.

Return values

A `String` containing the attribute's relationship type.

getTypeName()

Retrieves the name of the attribute's data type.

Syntax

```
public String getTypeName();
```

Parameters

None.

Return values

A `String` containing the name of the type of the attribute. See Table 151 on page 437 for a list of string attribute types.

getTypeNum()

Retrieves the numeric type code for the data type of an attribute.

Syntax

```
public String getTypeNum();
```

Parameters

None.

Return values

The numeric type code of the type of the attribute. See Table 151 on page 437 for a list of numeric attribute-type constants.

hasCardinality()

Determines if the attribute has the same cardinality as the cardinality value passed in as a parameter. This method is used to test cardinality of complex attributes (subobjects and containers). Valid cardinality values are from 1 to n.

Syntax

```
public boolean hasCardinality(String card);
```

Parameters

card Is the cardinality value to use for checking. Use one of the cardinality constants:

```
CxObjectAttr.MULTIPLECARDSTRING
CxObjectAttr.SINGLECARDSTRING
```

Return values

Returns True if the cardinality of the attribute matches the parameter value; otherwise, returns False.

hasName()

Determines if the name of the attribute matches the name passed in as a parameter.

Syntax

```
public boolean hasName(String name);
```

Parameters

name Is the name of the attribute passed in to the method.

Return values

Returns True if the attribute name matches the passed-in name; otherwise, returns False.

hasType()

Determines if the data type of the attribute matches the type name passed in as a parameter.

Syntax

```
public boolean hasType(String typeName);
```

Parameters

typeName Is the type of the attribute passed in to the method. Use one of the string attribute-type constants:

```
CxObjectAttr.BOOLSTRING
CxObjectAttr.DATESTRING
CxObjectAttr.DOUBSTRING
CxObjectAttr.FLTSTRING
CxObjectAttr.INTSTRING
CxObjectAttr.LONGTEXTSTRING
CxObjectAttr.STRSTRING
```

Return values

Returns True if the attribute type matches the passed-in type name; otherwise, returns False.

isForeignKeyAttr()

Determines if this attribute is a part of the foreign key of the business object.

Syntax

```
public boolean isForeignKeyAttr();
```

Parameters

None.

Return values

Returns True if the attribute is a foreign key, or part of the foreign key, for the business object; otherwise, returns False.

isKeyAttr()

Determines if this attribute is a part of the business object key set.

Syntax

```
public boolean isKeyAttr();
```

Parameters

None.

Return values

Returns True if the attribute is a key, or part of the key set, for the business object; otherwise, returns False.

isMultipleCard()

Determines if this attribute is a multiple cardinality.

Syntax

```
public boolean isMultipleCard();
```

Parameters

None.

Return values

Returns True if the attribute is a multiple cardinality; otherwise, returns False.

isObjectType()

Determines if this attribute's data type is an object type; that is, if it is a complex attribute (a container or a subobject).

Syntax

```
public boolean isObjectType();
```

Parameters

None.

Return values

Returns True if the attribute is an object type or complex attribute, such as a container or subobject; otherwise, returns False.

isRequiredAttr()

Determines if this attribute is a required attribute for the business object. If the attribute is required, it must have a value.

Syntax

```
public boolean isRequiredAttr();
```

Parameters

None.

Return values

Returns True if the attribute is required for the business object; otherwise, returns False.

isType()

Determines that the value passed in to the method is of the same type as that of the attribute.

Syntax

```
public boolean isType(Object value);
```

Parameters

value Is the type of the attribute to check against.

Return values

Returns True if the type of the attribute matches the passed-in type; otherwise, returns False.

Chapter 31. CxObjectContainerInterface interface

The CxObjectContainerInterface interface creates and maintains an array of one or more child business objects. It is part of the CxCommon package. This interface supports business objects with a hierarchical structure. Each object instance created from the CxObjectContainerInterface is a container object into which you can insert business objects of the same type. These inserted objects are instances of a business object definition referenced by a compound attribute of a parent business object. The inserted objects are child business objects in the hierarchy.

Note: The CWConnectorBusObj class is the Java connector library method that is a wrapper for the methods in the CxObjectContainerInterface interface of the low-level Java connector library. The CWConnectorBusObj class provides access to a business object, business object array, business object definition, and attributes. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

Table 153 summarizes the methods in the CxObjectContainerInterface interface.

Table 153. Member methods of the CxObjectContainerInterface interface

Member method	Description	Page
getBusinessObject()	Retrieves the child business object that occupies a specified position in a business object array.	445
getObjectCount()	Retrieves the number of child business objects in a business object array.	446
insertBusinessObject()	Inserts a child business object into a business object array at the next available position.	446
removeAllObjects()	Removes all business objects in a business object array.	447
removeBusinessObjectAt()	Removes the business object at the specified position in a business object array.	447
setBusinessObject()	Inserts a child business object into a business object array at a specified position.	447

Note: The deprecated name for an array of child business objects is a "business object container". This term is also used to name the connector library class that provides methods for accessing the child business objects in a business object array. You can think of this class as providing methods for handling an array of business objects.

getBusinessObject()

Retrieves the child business object that occupies a specified position in a business object array.

Syntax

```
public BusinessObjectInterface getBusinessObject(int index);
```

Parameters

index Is an integer that specifies the position of a child business object in a business object array.

Return values

The child business object, or `null` if there is no business object at the specified position in the business object array.

See also

See also the description of the `setBusinessObject()` method.

getObjectCount()

Retrieves the number of child business objects in a business object array.

Syntax

```
public int getObjectCount();
```

Parameters

None.

Return values

An integer that indicates the number of child business objects in a business object array.

Notes

You can use the `insertBusinessObject()` method to insert child business objects into the business object array.

See also

See also the description of the `insertBusinessObject()` method.

insertBusinessObject()

Inserts a child business object into a business object array at the next available position.

Syntax

```
public void insertBusinessObject(BusinessObjectInterface theChildBusObj);
```

Parameters

theChildBusObj Is the child business object to be inserted.

Return values

None.

Exceptions

`CxObjectInvalidAttrException`
Thrown if the passed-in business object is not the same type as the objects contained by the array.

See also

See also the description of the `setBusinessObject()` method.

removeAllObjects()

Removes all business objects in a business object array.

Syntax

```
public void removeAllObjects();
```

Parameters

None.

Return values

None.

removeBusinessObjectAt()

Removes a business object at a specified position in a business object array.

Syntax

```
public void removeBusinessObjectAt(int index);
```

Parameters

index Is an integer that specifies the position for a child business object in a business object array.

Return values

None.

Exceptions

`CxObjectNoSuchAttributeException`
Thrown if the position specified is not valid for this business object.

Notes

After the remove operation, the business object array is compacted. Indexes are decremented for all business objects that have an index number higher than that of the removed business object.

setBusinessObject()

Inserts a child business object into a business object array at a specified position.

Syntax

```
public BusinessObjectInterface setBusinessObject(int index,
    BusinessObjectInterface theChildBusObj);
```

Parameters

index Is an integer that specifies the position for a child business object in a business object array.

theChildBusObj Is a child business object.

Return values

The original business object, if one was replaced as a result of the insertion. Otherwise, returns `null`.

Exceptions

`CxObjectInvalidAttrException`
Thrown if the type of the passed-in business object attribute is not of the type that the business object array handles.

Notes

If there is already a business object at the specified position, the new one replaces it. The old one is deleted and returned to the caller.

See also

See also the description of the `getBusinessObject()` method.

Chapter 32. CxProperty class

The CxProperty class represents a hierarchical connector configuration property for a low-level Java connector. A hierarchical connector configuration property can contain one or more values and these values can be either string values or other (child) connector properties.

Note: The CWProperty class is the Java connector library method that is a wrapper for the CxProperty class of the low-level Java connector library. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

Table 154 summarizes the methods in the CxProperty class.

Table 154. Member methods of the CxProperty class

Member method	Description	Page
CxProperty()	Creates a connector-property object.	449
getAllChildProps()	Retrieves *all* child properties from the hierarchical connector configuration property.	450
getChildProp()	Retrieves *a specified* child property from the hierarchical connector configuration property., at any level in the property hierarchy.	451
getEncryptionFlag()	Retrieves the encryption flag for the connector configuration property.	452
getName()	Retrieves the name of the connector configuration property.	452
getStringValues()	Retrieves all string values from the hierarchical connector configuration property.	452
hasChildren()	Determines whether the connector configuration property has any child properties.	453
setEncryptionFlag()	Sets the encryption flag for the hierarchical connector configuration property.	454
setValues()	Sets the values of the hierarchical connector configuration property.	454

CxProperty()

Creates a hierarchical connector-property object.

Syntax

```
public CxProperty();
public CxProperty(String propName, String simplePropValue);
public CxProperty(String propName, Object[] hierPropValues);
public CxProperty(String propName, org.w3c.dom.Element xmlElement);
```

Parameters

propName Specifies the name of the connector configuration property.

simplePropValue
: Is a String value with which to initialize a simple connector property.

hierPropValues
: Is an array of connector-property (`CWProperty`) objects with which to initialize a hierarchical connector property.

xmlElement
: Is an XML `Element` object with which to initialize the connector property.

Return values

A `CxProperty` object containing the newly created hierarchical connector property.

Notes

The `CxProperty()` constructor provides the following forms:

- The first form creates an empty connector-property object. You can use other methods of the `CWProperty` class to populate this object.
- The second form creates a connector-property object for a simple connector property, with a property name and a string value that you specify.
- The third form creates a connector-property object for a hierarchical connector property, with a property name and array of hierarchical properties that you specify.
- The fourth form creates a connector-property object and an XML `Element` object.

---

getAllChildProps()

Retrieves *all* child properties for the hierarchical connector configuration property.

Syntax

```
public CxProperty[] getAllChildProps();
```

Parameters

None.

Return values

A reference to an array of `CxProperty` objects, each of which represents one connector property in the hierarchical connector property. If the hierarchical connector property does not contain any child properties, the method returns `null`.

Exceptions

None.

Notes

The `getAllChildProps()` method retrieves *all* child properties for the hierarchical connector configuration property. The retrieved properties are only those of the children of the current hierarchical property; they do *not* include any grandchildren, great-grandchildren, and so on. To retrieve child properties at lower levels in the hierarchy, you must first obtain the connector-property object for a property at a particular level and then use a method such as `getAllChildProps()` or `getChildProp()` to retrieve its children.

Note: You can use the `getChildProp()` to retrieve a specified child, grandchild, and so on down the property hierarchy.

To retrieve a specified child property, use the `getChildProp()` method. To retrieve all child properties with a specified prefix, you can use the `getChildPropsWithPrefix()` method.

See also

getChildProp()

---

getChildProp()

Retrieves a specified child property for the hierarchical connector configuration property at any level of the property hierarchy.

Syntax

```
public CxProperty getChildProp(String propName);
```

Parameters

propName Specifies the name of the connector configuration property to retrieve.

Return values

A `CxProperty` object that contains the retrieved property from the hierarchy. If the specified property does not exist in the current hierarchical connector property, the method returns `null`.

Exceptions

None.

Notes

The `getChildProp()` method retrieves the child property whose name matches *propName* from the hierarchical connector configuration property. You can retrieve a child property *at any level* of the current property hierarchical; you can specify a grandchild, great-grandchild, and so on. The *propName* of the retrieved child property has the form:

```
child/grandchild/great-grandchild/....
```

For example, suppose you have the property hierarchy shown in Figure 77 on page 394. To obtain the name of the port for `Listener1`, you must first retrieve the top-level connector-object for `ProtocolListener` (for example, into `topLevelProp`). You can then use the following call to retrieve the port name of `Listener1`:

```
CxProperty listenerPort = topLevelProp.getChildProp("Listener1/Port");
```

To retrieve *all* child properties at the top level of the hierarchy, you can use the `getAllChildProps()` method.

See also

getAllChildProps()

getEncryptionFlag()

Retrieves the encryption flag of the hierarchical connector configuration property from the connector-property object.

Syntax

```
public boolean getEncryptionFlag();
```

Parameters

None.

Return values

A Boolean value that indicates whether the current connector configuration property's value is encrypted.

Exceptions

None.

Notes

The `getEncryptionFlag()` method obtains the boolean encryption flag from the connector-property object. This flag indicates whether the connector property's string values are encrypted.

Note: In Connector Configurator, encrypted values display as a string of asterisk (*) characters.

See also

setEncryptionFlag()

getName()

Retrieves the name of the connector configuration property from the connector-property object.

Syntax

```
public String getName();
```

Parameters

None.

Return values

A String that contains the name of the connector configuration property.

Exceptions

None.

getStringValues()

Retrieves *all* string values for the hierarchical connector configuration property.

Syntax

```
public String[] getStringValues();
```

Parameters

None.

Return values

A reference to an array of `String` objects, each of which represents one string value for the hierarchical connector property. If the hierarchical connector property does not contain any string values, the method returns `null`.

Exceptions

None.

Notes

The `getStringValues()` method retrieves *all* string values for the hierarchical connector configuration property. The retrieved string values are only those of the current hierarchical property; they do *not* include any values in child properties. To retrieve string values at lower levels in the hierarchy, you can do either of the following:

- Use the `getChildPropValue()` method to retrieve the string value of a specified child property.
- Obtain the connector-property object for a property at a particular level and then use a method such as `getStringValues()` or `getChildPropValue()` to retrieve its string values.

Before a call to `getHierChildProps()`, you can use the `hasChildren()` method to verify that the hierarchical connector property has child properties. To retrieve child properties, use the `getChildProp()` or `getAllChildProps()` method.

See also

getChildPropValue(), getAllChildProps(), getChildProp(), setValues()

---

hasChildren()

Determines whether the current connector property contains any child properties.

Syntax

```
public boolean hasChildren();
```

Parameters

None.

Return values

A `boolean` that indicates whether the hierarchical connector property contains any child properties. The method returns `true` if it does contain child properties; otherwise, it returns `false`.

Exceptions

None.

Notes

The `hasChildren()` method is useful for determining which of the `CWProperty` methods to use to extract the value of a hierarchical connector property:

- If `hasChildren()` returns `true`, use one of the following value methods to retrieve child properties:

To obtain all child properties	`getHierChildProps()`
To obtain a specified child property	`getHierChildProp()`

- If `hasChildren()` returns `false`, use one of the following value methods to retrieve string values:

To obtain all string values	`getStringValues()`
To obtain the string values of a specified child property	`getChildPropValue()`

See also

getChildPropValue(), getHierChildProp(), getConnectorBOHandlerForBO(), getStringValues(),getVersion()

setEncryptionFlag()

Sets the encryption flag of a connector configuration property in its connector-property object.

Syntax

```
public void setEncryptionFlag(boolean encryptFlag);
```

Parameters

encryptFlag Is a boolean value to indicate whether the current connector configuration property's value should be encrypted.

Return values

None.

Notes

The `setEncryptionFlag()` method sets the boolean encryption flag from the connector-property object. This flag indicates whether the connector property's string values are encrypted.

Note: In Connector Configurator, encrypted values display as a string of asterisk (*) characters.

See also

getEncryptionFlag()

setValues()

Sets the values of the hierarchical connector configuration property.

Syntax

```
public void setValues(Object[] propValues);
```

Parameters

propValues Is an array of `Object` values, each array element is a single property value.

Return values

None.

Exceptions

None.

Notes

The `setValues()` method allows you to set the values of a hierarchical connector configuration property. You specify the property values in the *propValues* array, which is an array of `Objects`. Therefore, you can pass both string and child-property values in this single array. Make sure you assign property values in the *propValues* array in the order that they are defined within the hierarchical connector property.

For example, the following call to `setValues()` assigns both a string value and a child property to the connector property in `topLevelProp`:

```
Object[] propValues;
CxProperty childProp;

propValues[0] = "stringValue"
propValue[1] = childProp;
topLevelProp.setValues(propValues);
```

See also

getHierChildProp(), getConnectorBOHandlerForBO(), getStringValues()

Chapter 33. CxStatusConstants class

The CxStatusConstants class defines outcome-status constants for the low-level Java connector library. It is part of the CxCommon package.

Note: The CWConnectorConstant class is the Java connector library class that is a wrapper for the CxStatusConstants class of the low-level Java connector library. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

Outcome-status constants

Many methods of the low-level Java connector library return an integer outcome status to indicate the success of the method. Table 155 summarizes the static outcome-status constants, which are defined in the CxStatusConstants class.

Table 155. Outcome-status constants in the CxStatusConstants class

Constant name	Meaning
SUCCEED	The operation completed successfully.
FAIL	The operation failed for an unspecified reason.
APPRESPONSETIMEOUT	The application is not responding.
BO_DOES_NOT_EXIST	The requested business object does not exist.
CONNECTOR_NOT_ACTIVE	The connector is not active; that is, it is in the paused state.
MULTIPLE_HITS	The integration broker requested a retrieve-by-content but the connector found more than one matching record. The status indicates that more than one record matched the search requirements.
NO_SUBSCRIPTION_FOUND	Cannot find any subscriptions for the business object.
RETRIEVEBYCONTENT_FAILED	Retrieve by content failed.
UNABLETOLOGIN	Cannot login to the application.
VALCHANGE	The operation successfully completed and changed the value of the object in the target application.
VALDUPES	The requested operation failed because multiple records were found for the same key field (or fields).

Chapter 34. JavaConnectorUtil class

The JavaConnectorUtil class is a final class that contains miscellaneous utility methods for use in a low-level Java connector. It is part of the AppSide_Connector package. A connector developer can use these static methods for generating and logging messages and creating business objects.

Note: The CWConnectorUtil class is the Java connector library class that is a wrapper for the JavaConnectorUtil class of the low-level Java connector library. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

This class contains the following:

- "Static constants"
- "Methods"

Static constants

The JavaConnectorUtil class defines a number of static constants. See Table 156.

Table 156. Static constants defined in the JavaConnectorUtil class

Constant name	Meaning
Message-file constants	
CONNECTOR_MESSAGE_FILE	Use the connector message file to generate messages.
INFRASTRUCTURE_MESSAGE_FILE	Use the InterChange Server message file (InterchangeSystem.txt) to generate messages. **Important:** Connectors should *not* obtain messages from the InterchangeSystem.txt file. Instead, they should always use their local connector message file.
Message-type constants	
XRD_WARNING	A warning message
XRD_TRACE	A trace message
XRD_INFO	An informational message
XRD_ERROR	An error message
XRD_FATAL	A fatal error message
Trace-level constants	
LEVEL1	Level 1 of tracing
LEVEL2	Level 2 of tracing
LEVEL3	Level 3 of tracing
LEVEL4	Level 4 of tracing
LEVEL5	Level 5 of tracing

Methods

Table 157 summarizes the methods in the JavaConnectorUtil class.

Table 157. Member methods of the JavaConnectorUtil class

Member method	Description	Page
createBusinessObject()	Creates a business object.	460

Table 157. Member methods of the JavaConnectorUtil class (continued)

Member method	Description	Page
createContainer()	Creates a container	461
generateMsg()	Generates a message from a message file that you specify, depending on the trace level. You can optionally specify a trace level.	461
getAllConfigProp()	Retrieves all properties for the connector as hierarchical connector properties.	462
getAllConnectorAgentProperties()	Retrieves all properties for the connector.	463
getAllStandardProperties()	Retrieves all standard connector properties, as hierarchical connector properties.	463
getAllUserProperties()	Retrieves all connector-specific properties, as hierarchical connector properties.	464
getBlankValue()	Retrieves the special Blank attribute value.	464
getConfigProp()	Retrieves a property for the connector from the repository.	465
getEncoding()	Retrieves the character encoding that the connector framework is using.	465
getIgnoreValue()	Retrieves the special Ignore attribute value.	466
getLocale()	Retrieves the locale of the connector framework.	466
getOneConfigProp()	Retrieve a specified hierarchical connector property.	467
getSupportedBusObjNames()	Retrieve a list of supported business objects for the connector.	467
initAndValidateAttributes()	Initializes all required attributes to their default values.	468
isBlankValue()	Tests if a value equals the special Blank attribute value.	470
isIgnoreValue()	Tests if a value equals the special Ignore attribute value.	470
isTraceEnabled()	Tests if the trace level is greater than or equal to a specified trace level.	470
logMsg()	Logs a message. You can optionally send an email message if the message severity is set to error or fatal error	471
traceWrite()	Writes a trace message.	471

createBusinessObject()

Creates a new business object.

Syntax

```
public static BusinessObjectInterface createBusinessObject(
    String busObjName);

public static BusinessObjectInterface createBusinessObject(
    String busObjName, Locale localeObject);

public static BusinessObjectInterface createBusObj(
    String busObjName, String localeName);
```

Parameters

busObjName Is the name of the business object to create.

localeObject
Is the Java `Locale` object that identifies the locale to associate with the business object.

localeName
Is the name of the locale to associate with the business object.

Return values

A `BusinessObjectInterface` object containing the newly created business object.

Exceptions

`BusObjSpecNameNotFoundException`
Thrown when the business object specification is not found for the name specified.

Notes

The `createBusinessObject()` method creates a new business object instance whose type is the business object definition you specify in *name*. If you specify a *localeObject* or *localeName*, this locale applies to the data in the business object, not to the name of the business object definition or its attributes (which must be characters in the code set associated with the U.S. English locale, `en_US`). For a description of the format for *localeName*, see "Design Considerations for an Internationalized Connector," on page 54.

createContainer()

Creates an instance of a business object array (container).

Syntax

```
public static CxObjectContainerInterface createContainer(String name);
```

Parameters

name Specifies the name of the business object container to create.

Return values

A `CxObjectContainerInterface` object containing the newly created business object.

Exceptions

`BusObjSpecNameNotFoundException`
Thrown when the business object specification is not found for the name specified.

generateMsg()

Generates a message from a set of predefined messages in a message file.

Syntax

```
public final static String generateMsg(int traceLevel, int msgNum,
    int msgType, int isGlobal, int argCount, Vector msgParams);

public final static String generateMsg(int msgNum, int msgType,
    int isGlobal, int argCount, Vector msgParams);
```

Parameters

traceLevel specifies the trace level at which to generate the message. When this parameter is omitted, the method generates the message regardless of the trace level. The message is generated only if the *traceLevel* value is equal to or less than the current trace level of the connector.

msgNum specifies the message number (identifier) in the message file.

msgType is one of the following message types:

```
JavaConnectorUtil.XRD_WARNING
JavaConnectorUtil.XRD_ERROR
JavaConnectorUtil.XRD_FATAL
JavaConnectorUtil.XRD_INFO
JavaConnectorUtil.XRD_TRACE
```

isGlobal is the `CONNECTOR_MESSAGE_FILE` message-file constant defined in the `CWConnectorLogAndTrace` class to indicate that the message file is the connector message file.

argCount is an integer that specifies the number of parameters within the message text. To determine the number, refer to the message in the message file.

msgParams is a list of parameters for the message text.

Return values

A `String` containing the generated message, or `null` if the trace level is greater than the trace level of the connector.

Notes

The `generateMsg()` method provides two forms:

- Use the first form of the method (where *traceLevel* is the first parameter) for tracing messages. For the message to be generated, the trace level must be less than or equal to the trace level of the connector. You then use the `traceWrite()` method to send the trace message to the log destination.
- Use the second form of the signature (where *msgNum* is the first parameter) for logging. You then use the `logMsg()` method to send the log message to the log destination.

---

getAllConfigProp()

Retrieves a list of all configuration properties for the current connector as hierarchical connector properties.

Syntax

```
public static CxProperty[] getAllConfigProp();
```

Parameters

None.

Return values

A reference to an array of `CxProperty` objects, each of which contains one connector property for the current connector.

Notes

The `getAllConfigProp()` method retrieves the connector configuration properties as an array of `CxProperty` objects. Each connector-property (`CxProperty`) object contains a single connector property and can hold a single value, another property, or a combination of values and child properties. Use methods of the `CxProperty` class (such as `getAllChildProps()` and `getStringValues()`) to obtain the values from a connector-property object.

See also

getConfigProp(), getAllConnectorAgentProperties(), getAllStandardProperties(), getAllUserProperties()

---

getAllConnectorAgentProperties()

Retrieves a list of all configuration properties for the current connector.

Syntax

```
public static Hashtable getAllConnectorAgentProperties();
```

Parameters

None.

Return values

A reference to a `Hashtable` object that contains the connector properties for the current connector.

Notes

The `getAllConnectorAgentProperties()` method retrieves the connector configuration properties as a Java `Hashtable` object, which maps keys to values. The keys are the names of the properties and values are the associated property values. Use methods of the `Hashtable` structure (such as `keys()` and `elements()`) to obtain the information from this structure.

Examples

```
Hashtable ht = new Hastable();
ht = JavaConnectorUtil.getAllConnectorAgentProperties();
int size = ht.size();
Enumeration properties = ht.keys();
Enumeration values = ht.elements();

while (properties.hasMoreElements()) {
    System.out.print((String)properties.nextElement());
    System.out.print("=");
    System.out.println((String)values.nextElement());
    System.out.println("Property set");
}
```

---

getAllStandardProperties()

Retrieves all the connector's standard properties.

Syntax

```
static CxProperty[] getAllStandardProperties();
```

Parameters

None.

Return values

A reference to an array of `CxProperty` objects, each of which contains one standard connector property for the current connector.

Notes

The `getAllStandardProperties()` method retrieves the standard connector configuration properties as an array of `CxProperty` objects. Each connector-property (`CxProperty`) object contains a single standard connector property and can hold a single value, another property, or a combination of values and child properties. Use methods of the `CxProperty` class (such as `getAllChildProps()` and `getStringValues()`) to obtain the values from a connector-property object.

See also

getAllConfigProp(), getAllUserProperties(), getOneConfigProp()

---

getAllUserProperties()

Retrieves all the connector's connector-specific properties.

Syntax

```
static CxProperty[] getAllUserProperties();
```

Parameters

None.

Return values

A reference to an array of `CxProperty` objects, each of which contains one connector-specific connector property for the current connector.

Notes

The `getAllUserProperties()` method retrieves the connedtor-specific configuration properties as an array of `CxProperty` objects. Each connector-property (`CxProperty`) object contains a single connector-specific property and can hold a single value, another property, or a combination of values and child properties. Use methods of the `CxProperty` class (such as `getAllChildProps()` and `getStringValues()`) to obtain the values from a connector-property object.

See also

getAllConfigProp(), getAllStandardProperties(), getOneConfigProp()

---

getBlankValue()

Retrieves the special Blank attribute value.

Syntax

```
public static String getBlankValue();
```

Parameters

None.

Return values

A `String` object containing the special Blank attribute value.

getConfigProp()

Retrieves the configuration property value for a connector from the repository.

Syntax

```
public final static String getConfigProp(String propName);
```

Parameters

propName Is the name of the property to retrieve.

Return values

A `String` object containing the property value. If the property name is not found, the method returns `null`.

Notes

When you call `getConfigProp("ConnectorName")` in a parallel-process connector (one that has the `ParallelProcessDegree` connector property set to a value greater than 1), the method always returns the name of the connector-agent master process, regardless of whether it is called in the master process or a slave process.

getEncoding()

Retrieves the character encoding that the connector framework is using.

Syntax

```
public String getEncoding();
```

Parameters

None.

Return values

A `String` object containing the connector framework's character encoding.

Notes

The `getEncoding()` method retrieves the connector framework's character encoding, which is part of the locale. The locale specifies cultural conventions for data according to language, country (or territory), and a character encoding. The connector framework's character encoding should indicate the character encoding of the connector application. The connector framework's character encoding using the following hierarchy:

- The `CharacterEncoding` connector configuration property in the repository

WebSphere InterChange Server

If a local configuration file exists, the setting of the `CharacterEncoding` connector configuration property in this local file takes precedence. If no local configuration file exists, the setting of the `CharacterEncoding` property is one from the set of connector configuration properties downloaded from the InterChange Server repository at connector startup.

- The character encoding from the Java environment, which Unicode (UCS-2)

This method is useful when the connector needs to perform character-encoding processing, such as character conversion.

See also

getGlobalLocale()

getIgnoreValue()

Retrieves the value for the special Ignore attribute value

Syntax

```
public static String getIgnoreValue();
```

Parameters

None.

Return values

A `String` object containing the special "Ignore" attribute value.

getLocale()

Retrieves the locale of the connector framework.

Syntax

```
public String getLocale();
```

Parameters

None.

Return values

A `String` object containing the connector framework's locale setting.

Notes

The `getLocale()` method retrieves the connector framework's locale, which defines cultural conventions for data according to language, country (or territory), and a character encoding. The connector framework's locale should indicate the locale of the connector application. The connector framework's locale is set using the following hierarchy:

- The `LOCALE` connector configuration property in the repository

WebSphere InterChange Server

If a local configuration file exists, the setting of the `Locale` connector configuration property in this local file takes precedence. If no local configuration file exists, the setting of the `Locale` property is the one from the set of connector configuration properties downloaded from the InterChange Server repository at connector startup.

- The locale from the Java environment, which is the locale from the operating system

This method is useful when the connector needs to perform locale-sensitive processing.

See also

createBusinessObject(), getGlobalEncoding(), getLocale() (within the `BusinessObjectInterface` interface)

getOneConfigProp()

Retrieves the top-level connector-object for a specified hierarchical connector configuration property.

Syntax

```
public static CxProperty getOneConfigProp(String propName);
```

Parameters

propName Is the name of the hierarchical connector property to retrieve.

Return values

A `CxProperty` object that contains the top-level connector-property object for the specified hierarchical connector property. If the property name is not found, the method returns `null`.

Notes

The `getOneConfigProp()` method obtains the top-level connector-property (`CxProperty`) object. From this retrieved object, you can use methods of the `CxProperty` class to obtain the desired values of the connector property.

Note: Values of connector configuration properties are downloaded to the connector during its initialization. If you specify a *propName* for a connector property that has not been downloaded, `getOneConfigProp()` returns `null`.

See also

getAllConfigProp(), getConfigProp()

getSupportedBusObjNames()

Retrieves a list of supported business objects for the current connector.

WebSphere InterChange Server
This method is only valid when the integration broker is InterChange Server.

Syntax

```
public static String[] getSupportedBusObjNames()
```

Parameters

None.

Return values

A `String` array that contains a list of the names of the supported business objects for the connector.

Notes

The `getSupportedBusObjNames()` method returns a list of top-level supported business objects for the current connector; that is, if the connector supports business objects that contain child business objects, `getSupportedBusObjNames()` includes only the name of the parent object in its list.

Note: The `getSupportedBusObjNames()` method is *only* supported when the connector is using a version 4.0 or later InterChange Server as its integration broker.

initAndValidateAttributes()

Initializes attributes that do not have values set, but are marked as required, with their default values.

Syntax

```
public static void initAndValidateAttributes(
    BusinessObjectInterface theBusObj);
```

Parameters

theBusObj Is the business object whose attributes this method sets.

Return values

None.

Exceptions

`BusObjSpecNameNotFoundException`
Thrown when the name of the specified business object does not match any of the business object definitions in the repository.

`SetDefaultFailedException`
Thrown when the attribute's default value could not be set and there is no default value specified for the attribute in the business object definition.

Notes

The initAndValidateAttributes() method has two purposes:

- It *initializes* attributes by setting the default value for each attribute under the following conditions:
  - When the UseDefaults connector configuration property is set to true
  - When the attribute's isRequired property is set to true
  - When the attribute's value is *not* currently set (has the special Ignore value of CxIgnore)
  - When the attribute's Default Value property specifies a default value
- It *validates* attributes by throwing a SetDefaultFailedException exception under the following conditions:
  - When the attribute's isRequired property is set to true
  - When the attribute does *not* have a Default Value property that defines its default value

  In case of failure, no value exists some attributes (those without default values) after initAndValidateAttributes() finishes default-value processing . You might want to code your connector's application-specific component to catch this exception and return CxStatusConstants.FAIL.

The initAndValidateAttributes()method looks at every attribute in all levels of a business object and determines the following:

- Whether an attribute is required
- Whether the attribute has a value in the business object instance
- Whether the UseDefaults configuration property is set to true

If an attribute is required and UseDefaults is true, initAndValidateAttributes() sets the value of any unset attribute to its default value. To have initAndValidateAttributes() set the attribute value to the special Blank value (CxBlank), you can set the attribute's default value to the string "CxBlank". If the attribute does *not* have a default value, initAndValidateAttributes() throws the SetDefaultFailedException exception.

Note: If an attribute is a key or other attribute whose value is generated by the application, the business object definition should not provide default values, and the Required property for the attribute should be set to false.

The initAndValidateAttributes() method is usually called from the business-object-handler doVerbFor() method to ensure that required attributes have values before a Create operation is performed in an application. In the doVerbFor() method, you can call the initAndValidateAttributes() method for the Create verb. You can also call it for the Update verb, before it performs a Create.

To use initAndValidateAttributes(), you must also do the following:

- Design business objects so that the IsRequired property is set to true for required attributes and that required attributes have default values specified in their Default Value property.
- Add the UseDefaults connector configuration property to the list of connector-specific properties for the connector. Set this property to true.

isBlankValue()

Determines if an attribute value is the special Blank (CxBlank) attribute value.

Syntax

```
public static boolean isBlankValue(Object value);
```

Parameters

value Is the value to compare with the special Blank value.

Return values

Returns True if the value is the special Blank attribute value; otherwise, returns False.

isIgnoreValue()

Determines if an attribute value is the special Ignore (CxIgnore) attribute value, which signifies that this attribute is to be ignored while processing the business object.

Syntax

```
public static boolean isIgnoreValue(Object value);
```

Parameters

value Is the value to compare with the special Ignore value.

Return values

Returns True if the value is equal to the special Ignore attribute value; otherwise, returns False.

isTraceEnabled()

Determines if the trace level is greater than or equal to the trace level for which it is looking, if tracing is enabled at this level.

Syntax

```
public final static boolean isTraceEnabled(int traceLevel);
```

Parameters

traceLevel Is the trace level to check.

Return values

Returns True if the agent trace level is greater than or equal to the trace level passed in.

Notes

Use this method before generating a message.

logMsg()

Logs a message to the connector's log destination.

Syntax

```
public final static void logMsg(String msg);
public final static void logMsg(String msg, int severity);
```

Parameters

msg Is the message text to be logged.

severity Is one of the following message types:

```
JavaConnectorUtil.XRD_WARNING
                        JavaConnectorUtil.XRD_ERROR
                        JavaConnectorUtil.XRD_FATAL
                        JavaConnectorUtil.XRD_INFO
JavaConnectorUtil.XRD_TRACE
```

Return values

None.

Notes

The `logMsg()` method sends the specified *msg* text to the log destination. You establish the name of a connector's log destination through the Logging section in the Trace/Log File tab of Connector Configurator.

IBM recommends that log messages be contained in a message file and extracted with the `generateMsg()` method. This message file should be the connector message file, which contains messages specific to your connector. The `generateMsg()` method generates the message string for `logMsg()`. It retrieves a predefined message from a message file, formats the text, and returns a generated message string.

WebSphere InterChange Server

If `severity` is `XRD_ERROR` or `XRD_FATAL` and the connector configuration property `LogAtInterchangeEnd` is set, the error message is logged and an email notification is sent when email notification is on. See the *System Administration Guide* in the IBM WebSphere InterChange Server documentation set for information on how to set up email notification for errors.

Connector messages logged with `logMsg()` are viewable using LogViewer if the message strings were generated with `generateMsg()`.

See also

See the description of the `generateMsg()` method.

traceWrite()

Writes a trace message to the connector's trace destination.

Syntax

```
public final static void traceWrite(int traceLevel, String msg);
```

Parameters

traceLevel Is one of the following trace levels:

```
JavaConnectorUtil.LEVEL1
JavaConnectorUtil.LEVEL2
JavaConnectorUtil.LEVEL3
JavaConnectorUtil.LEVEL4
JavaConnectorUtil.LEVEL5
```

The method writes the trace message when the current trace level is greater than or equal to *traceLevel*.

Note: Do not specify a trace level of zero (`LEVEL0`) with a tracing message. A trace level of zero indicates that tracing is turned off. Therefore, any trace message associated with a *traceLevel* of `LEVEL0` will never print.

msg Is the message text to use for the trace message.

Return values

None.

Notes

You can use the `traceWrite()` method to write your own trace messages for a connector. Tracing is turned on for connectors when the `TraceLevel` connector configuration property is set to a nonzero value (any trace-level constant *except* `LEVEL0`).

The `traceWrite()` method sends the specified *msg* text to the trace destination when the current trace level is greater than or equal to *traceLevel*. You establish the name of a connector's trace destination through the Tracing section in the Trace/Log File tab of Connector Configurator.

Because trace messages are usually needed only during debugging, whether trace messages are contained in a message file is left at the discretion of the developer:

- If non-English-speaking users need to view trace messages, you need to internationalize these messages. Therefore, you must put the trace messages in a message file and extract them with the `generateMsg()` method. This message file should be the connector message file, which contains message specific to your connector. The `generateMsg()` method generates the message string for `traceWrite()`. It retrieves a predefined trace message from a message file, formats the text, and returns a generated message string.
- If only English-speaking users need to view trace messages, you do not need to internationalize these messages. Therefore, you can include the trace message (in English) directly in the call to `traceWrite()`. You do *not* need to use the `generateMsg()` method.

Connector messages logged with `traceWrite()` are *not* viewable using LogViewer.

See also

See the description of the `generateMsg()` method.

Chapter 35. ReturnStatusDescriptor class

The ReturnStatusDescriptor class enables low-level Java connectors to return error and informational messages in a return-status descriptor. It is part of the CxCommon package. This return-status descriptor provides additional status information is usually returned as part of the request response sent to the integration broker.

Note: The CWConnectorReturnStatusDescriptor class is the Java connector library class that is a wrapper for the ReturnStatusDescriptor class of the low-level Java connector library. Most Java-connector development should use the Java connector library. For more information on the classes of the Java connector library, see Chapter 9, "Overview of the Java connector library," on page 233.

WebSphere InterChange Server

If your business integration system uses InterChange Server, the connector framework returns the return-status descriptor to the collaboration that initiated the request. The collaboration can access the information in this return-status descriptor to obtain the status of its service call request.

Table 158 summarizes the methods in the ReturnStatusDescriptor class.

Table 158. Member methods of the ReturnStatusDescriptor class

Member method	Description	Page
getErrorString()	Retrieves the error message from the object.	473
getStatus()	Retrieves the status of the requested operation.	473
setErrorString()	Sets the error message into the object.	474
setStatus()	Sets the status of the requested operation.	474

getErrorString()

Retrieves a message string from a return-status descriptor. The message may be an error message or an informational message.

Syntax

```
public String getErrorString();
```

Parameters

None.

Return values

A String containing an error or informational message for the integration broker, or null.

getStatus()

Retrieves the status of the requested operation.

Syntax

```
public int getStatus();
```

Parameters

None.

Return values

An `int` value signaling the status of an operation.

setErrorString()

Sets the error or informational message into the `ReturnStatusDescriptor` object.

Syntax

```
public void setErrorString(String errorStr);
```

Parameters

errorStr Is the message string.

Return values

None.

setStatus()

Sets the status of the requested operation.

Syntax

```
public void setStatus(int status);
```

Parameters

status Is the status value.

Return values

None.

Chapter 36. Low-level Java exceptions

The exceptions in the low-level Java connector library are subclasses derived from an internal IBM WebSphere business integration system exception class. These subclasses represent an *exception object*, which methods of the low-level Java connector library can throw.

Note: The reference description for most low-level Java connector library methods lists the exceptions thrown by that method in the Exceptions section.

The low-level Java connector library exceptions provide the following:

- "Exception subclasses"
- "Methods"

Exception subclasses

This chapter lists the exception subclasses for the low-level Java connector library.

Table 159. Low-level Java connector library exceptions

Exception name	Definition
`BusObjInvalidVerbException`	Thrown when the specified verb is not supported by the business object.
`BusObjSpecNameNotFoundException`	Thrown when the specification for creating a business object cannot be found.
`CxObjectInvalidAttrException`	Thrown when the data type of the specified attribute does not match the data type that the attribute is defined to hold.
`CxObjectNoSuchAttributeException`	Thrown when the specified position or name of an attribute does not match the attribute name or attribute position within the existing business object.
`SetDefaultFailedException`	Thrown when setting a default value fails.

Methods

Table 160 summarizes the methods in the exception subclasses of the low-level Java connector library.

Table 160. Member methods of the Java exception subclasses

Member method	Description	Page
getFormattedMessage()	Formats the exception's message.	475

getFormattedMessage()

Formats the exception's message into a special format.

Syntax

```
public String getFormattedMessage();
```

Parameters

None.

Return values

A `String` containing the formatted error message, or `null` if the exception object is initialized with `null` when it is constructed.

Notes

The message that `getFormattedMessage()` returns has the following format:

`[Type: <MsgType>][MsgID: <msgId>][Mesg: <msg>]`

To retrieve the error message embedded in the exception, use the `getFormattedMessage()` method.

Part 5. Appendixes

Appendix A. Standard configuration properties for connectors

This appendix describes the standard configuration properties for the connector component of WebSphere Business Integration adapters. The information covers connectors running on the following integration brokers:

- WebSphere InterChange Server (ICS)
- WebSphere MQ Integrator, WebSphere MQ Integrator Broker, and WebSphere Business Integration Message Broker, collectively referred to as the WebSphere Message Brokers (WMQI).
- WebSphere Application Server (WAS)

Not every connector makes use of all these standard properties. When you select an integration broker from Connector Configurator, you will see a list of the standard properties that you need to configure for your adapter running with that broker.

For information about properties specific to the connector, see the relevant adapter user guide.

Note: In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes and follow the conventions for each operating system.

New and deleted properties

These standard properties have been added in this release.

New properties

- XMLNameSpaceFormat

Deleted properties

- RestartCount

Configuring standard connector properties

Adapter connectors have two types of configuration properties:

- Standard configuration properties
- Connector-specific configuration properties

This section describes the standard configuration properties. For information on configuration properties specific to a connector, see its adapter user guide.

Using Connector Configurator

You configure connector properties from Connector Configurator, which you access from System Manager. For more information on using Connector Configurator, refer to the Connector Configurator appendix.

Note: Connector Configurator and System Manager run only on the Windows system. If you are running the connector on a UNIX system, you must have a Windows machine with these tools installed. To set connector properties

for a connector that runs on UNIX, you must start up System Manager on the Windows machine, connect to the UNIX integration broker, and bring up Connector Configurator for the connector.

Setting and updating property values

The default length of a property field is 255 characters.

The connector uses the following order to determine a property's value (where the highest number overrides other values):

1. Default
2. Repository (only if WebSphere InterChange Server is the integration broker)
3. Local configuration file
4. Command line

A connector obtains its configuration values at startup. If you change the value of one or more connector properties during a run-time session, the property's **Update Method** determines how the change takes effect. There are four different update methods for standard connector properties:

- **Dynamic**
  The change takes effect immediately after it is saved in System Manager. If the connector is working in stand-alone mode (independently of System Manager), for example with one of the WebSphere message brokers, you can only change properties through the configuration file. In this case, a dynamic update is not possible.
- Component restart
  The change takes effect only after the connector is stopped and then restarted in System Manager. You do not need to stop and restart the application-specific component or the integration broker.
- Server restart
  The change takes effect only after you stop and restart the application-specific component and the integration broker.
- Agent restart (ICS only)
  The change takes effect only after you stop and restart the application-specific component.

To determine how a specific property is updated, refer to the **Update Method** column in the Connector Configurator window, or see the Update Method column in the Property Summary table below.

Summary of standard properties

Table 161 on page 481 provides a quick reference to the standard connector configuration properties. Not all the connectors make use of all these properties, and property settings may differ from integration broker to integration broker, as standard property dependencies are based on `RepositoryDirectory`.

You must set the values of some of these properties before running the connector. See the following section for an explanation of each property.

Table 161. Summary of standard configuration properties

Property name	Possible values	Default value	Update method	Notes
AdminInQueue	Valid JMS queue name	*CONNECTORNAME* /ADMININQUEUE	Component restart	Delivery-Transport is JMS
AdminOutQueue	Valid JMS queue name	*CONNECTORNAME*/ADMINOUTQUEUE	Component restart	Delivery-Transport is JMS
AgentConnections	1-4	1	Component restart	Delivery-Transport is MQ or IDL: Repository-Directory is <REMOTE>
AgentTraceLevel	0-5	0	Dynamic	
ApplicationName	Application name	Value specified for the connector application name	Component restart	
BrokerType	ICS, WMQI, WAS			
CharacterEncoding	ascii7, ascii8, SJIS, Cp949, GBK, Big5, Cp297, Cp273, Cp280, Cp284, Cp037, Cp437 **Note:** This is a subset of supported values.	ascii7	Component restart	
ConcurrentEventTriggeredFlows	1 to 32,767	1	Component restart	Repository-Directory is <REMOTE>
ContainerManagedEvents	No value or JMS	No value	Component restart	Delivery-Transport is JMS
ControllerStoreAndForwardMode	true or false	True	Dynamic	Repository-Directory is <REMOTE>
ControllerTraceLevel	0-5	0	Dynamic	Repository-Directory is <REMOTE>
DeliveryQueue		*CONNECTORNAME*/DELIVERYQUEUE	Component restart	JMS transport only
DeliveryTransport	MQ, IDL, or JMS	JMS	Component restart	If Repository-Directory is local, then value is JMS only
DuplicateEventElimination	True or False	False	Component restart	JMS transport only: Container-ManagedEvents must be <NONE>
FaultQueue		*CONNECTORNAME*/FAULTQUEUE	Component restart	JMS transport only
jms.FactoryClassName	CxCommon.Messaging.jms .IBMMQSeriesFactory or CxCommon.Messaging .jms.SonicMQFactory or any Java class name	CxCommon.Messaging. jms.IBMMQSeriesFactory	Component restart	JMS transport only

Table 161. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
jms.MessageBrokerName	If FactoryClassName is IBM, use crossworlds.queue.manager. If FactoryClassName is Sonic, use localhost:2506.	crossworlds.queue.manager	Component restart	JMS transport only
jms.NumConcurrentRequests	Positive integer	10	Component restart	JMS transport only
jms.Password	Any valid password		Component restart	JMS transport only
jms.UserName	Any valid name		Component restart	JMS transport only
JvmMaxHeapSize	Heap size in megabytes	128m	Component restart	Repository-Directory is <REMOTE>
JvmMaxNativeStackSize	Size of stack in kilobytes	128k	Component restart	Repository-Directory is <REMOTE>
JvmMinHeapSize	Heap size in megabytes	1m	Component restart	Repository-Directory is <REMOTE>
ListenerConcurrency	1- 100	1	Component restart	Delivery-Transport must be MQ
Locale	en_US, ja_JP, ko_KR, zh_CN, zh_TW, fr_FR, de_DE, it_IT, es_ES, pt_BR **Note:** This is a subset of the supported locales.	en_US	Component restart	
LogAtInterchangeEnd	True or False	False	Component restart	Repository-Directory must be <REMOTE>
MaxEventCapacity	1-2147483647	2147483647	Dynamic	Repository-Directory must be <REMOTE>
MessageFileName	Path or filename	InterchangeSystem.txt	Component restart	
MonitorQueue	Any valid queue name	*CONNECTORNAME*/MONITORQUEUE	Component restart	JMS transport only: DuplicateEvent-Elimination must be True
OADAutoRestartAgent	True or False	False	Dynamic	Repository-Directory must be <REMOTE>
OADMaxNumRetry	A positive number	1000	Dynamic	Repository-Directory must be <REMOTE>
OADRetryTimeInterval	A positive number in minutes	10	Dynamic	Repository-Directory must be <REMOTE>
PollEndTime	HH:MM	HH:MM	Component restart	

Table 161. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
PollFrequency	A positive integer in milliseconds no (to disable polling) key (to poll only when the letter p is entered in the connector's Command Prompt window)	10000	Dynamic	
PollQuantity	1-500	1	Agent restart	JMS transport only: Container-ManagedEvents is specified
PollStartTime	HH:MM(HH is 0-23, MM is 0-59)	HH:MM	Component restart	
RepositoryDirectory	Location of metadata repository		Agent restart	For ICS: set to <REMOTE>; For WebSphere MQ message brokers and WAS: set to C:\crossworlds\repository
RequestQueue	Valid JMS queue name	*CONNECTORNAME*/REQUESTQUEUE	Component restart	Delivery-Transport is JMS
ResponseQueue	Valid JMS queue name	*CONNECTORNAME*/RESPONSEQUEUE	Component restart	Delivery-Transport is JMS: required only if Repository-Directory is <REMOTE>
RestartRetryCount	0-99	3	Dynamic	
RestartRetryInterval	A sensible positive value in minutes: 1 - 2147483547	1	Dynamic	
RHF2MessageDomain	mrm, xml	mrm	Component restart	Only if Delivery-Transport is JMS and WireFormat is CwXML.
SourceQueue	Valid WebSphere MQ name	*CONNECTORNAME*/SOURCEQUEUE	Agent restart	Only if Delivery-Transport is JMS and Container-ManagedEvents is specified
SynchronousRequestQueue		*CONNECTORNAME*/SYNCHRONOUSREQUESTQUEUE	Component restart	Delivery-Transport is JMS
SynchronousRequestTimeout	0 - any number (millisecs)	0	Component restart	Delivery-Transport is JMS

Table 161. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
SynchronousResponseQueue		*CONNECTORNAME*/SYNCHRONOUSRESPONSEQUEUE	Component restart	Delivery-Transport is JMS
WireFormat	CwXML, CwBO	CwXML	Agent restart	CwXML if Repository-Directory is not <REMOTE>: CwBO if Repository-Directory is <REMOTE>
WsifSynchronousRequest Timeout	0 - any number (millisecs)	0	Component restart	WAS only
XMLNameSpaceFormat	short, long	short	Agent restart	WebSphere MQ message brokers and WAS only

Standard configuration properties

This section lists and defines each of the standard connector configuration properties.

AdminInQueue

The queue that is used by the integration broker to send administrative messages to the connector.

The default value is CONNECTORNAME/ADMININQUEUE.

AdminOutQueue

The queue that is used by the connector to send administrative messages to the integration broker.

The default value is CONNECTORNAME/ADMINOUTQUEUE.

AgentConnections

Applicable only if RepositoryDirectory is <REMOTE>.

The AgentConnections property controls the number of ORB connections opened by orb.init[].

By default, the value of this property is set to 1. There is no need to change this default.

AgentTraceLevel

Level of trace messages for the application-specific component. The default is 0. The connector delivers all trace messages applicable at the tracing level set or lower.

ApplicationName

Name that uniquely identifies the connector's application. This name is used by the system administrator to monitor the WebSphere business integration system environment. This property must have a value before you can run the connector.

BrokerType

Identifies the integration broker type that you are using. The options are ICS, WebSphere message brokers (WMQI, WMQIB or WBIMB) or WAS.

CharacterEncoding

Specifies the character code set used to map from a character (such as a letter of the alphabet, a numeric representation, or a punctuation mark) to a numeric value.

Note: Java-based connectors do not use this property. A C++ connector currently uses the value `ascii7` for this property.

By default, a subset of supported character encodings only is displayed in the drop list. To add other supported values to the drop list, you must manually modify the `\Data\Std\stdConnProps.xml` file in the product directory. For more information, see the appendix on Connector Configurator.

ConcurrentEventTriggeredFlows

Applicable only if `RepositoryDirectory` is <REMOTE>.

Determines how many business objects can be concurrently processed by the connector for event delivery. Set the value of this attribute to the number of business objects you want concurrently mapped and delivered. For example, set the value of this property to `5` to cause five business objects to be concurrently processed. The default value is `1`.

Setting this property to a value greater than `1` allows a connector for a source application to map multiple event business objects at the same time and deliver them to multiple collaboration instances simultaneously. This speeds delivery of business objects to the integration broker, particularly if the business objects use complex maps. Increasing the arrival rate of business objects to collaborations can improve overall performance in the system.

To implement concurrent processing for an entire flow (from a source application to a destination application), you must:

- Configure the collaboration to use multiple threads by setting its `Maximum number of concurrent events` property high enough to use multiple threads.
- Ensure that the destination application's application-specific component can process requests concurrently. That is, it must be multi-threaded, or be able to use connector agent parallelism and be configured for multiple processes. Set the `Parallel Process Degree` configuration property to a value greater than `1`.

The `ConcurrentEventTriggeredFlows` property has no effect on connector polling, which is single-threaded and performed serially.

ContainerManagedEvents

This property allows a JMS-enabled connector with a JMS event store to provide guaranteed event delivery, in which an event is removed from the source queue and placed on the destination queue as a single JMS transaction.

The default value is No value.

When ContainerManagedEvents is set to JMS, you must configure the following properties to enable guaranteed event delivery:

- PollQuantity = 1 to 500
- SourceQueue = CONNECTORNAME/SOURCEQUEUE

You must also configure a data handler with the MimeType, DHClass, and DataHandlerConfigMOName (optional) properties. To set those values, use the **Data Handler** tab in Connector Configurator. The fields for the values under the Data Handler tab will be displayed only if you have set ContainerManagedEvents to JMS.

Note: When ContainerManagedEvents is set to JMS, the connector does *not* call its pollForEvents() method, thereby disabling that method's functionality.

This property only appears if the DeliveryTransport property is set to the value JMS.

ControllerStoreAndForwardMode

Applicable only if RepositoryDirectory is <REMOTE>.

Sets the behavior of the connector controller after it detects that the destination application-specific component is unavailable.

If this property is set to true and the destination application-specific component is unavailable when an event reaches ICS, the connector controller blocks the request to the application-specific component. When the application-specific component becomes operational, the controller forwards the request to it.

However, if the destination application's application-specific component becomes unavailable **after** the connector controller forwards a service call request to it, the connector controller fails the request.

If this property is set to false, the connector controller begins failing all service call requests as soon as it detects that the destination application-specific component is unavailable.

The default is true.

ControllerTraceLevel

Applicable only if RepositoryDirectory is <REMOTE>.

Level of trace messages for the connector controller. The default is 0.

DeliveryQueue

Applicable only if DeliveryTransport is JMS.

The queue that is used by the connector to send business objects to the integration broker.

The default value is CONNECTORNAME/DELIVERYQUEUE.

DeliveryTransport

Specifies the transport mechanism for the delivery of events. Possible values are MQ for WebSphere MQ, IDL for CORBA IIOP, or JMS for Java Messaging Service.

- If ICS is the broker type, the value of the DeliveryTransport property can be MQ, IDL, or JMS, and the default is IDL.
- If the RepositoryDirectory is a local directory, the value may only be JMS.

The connector sends service call requests and administrative messages over CORBA IIOP if the value configured for the DeliveryTransport property is MQ or IDL.

WebSphere MQ and IDL

Use WebSphere MQ rather than IDL for event delivery transport, unless you must have only one product. WebSphere MQ offers the following advantages over IDL:

- Asynchronous communication:
  WebSphere MQ allows the application-specific component to poll and persistently store events even when the server is not available.
- Server side performance:
  WebSphere MQ provides faster performance on the server side. In optimized mode, WebSphere MQ stores only the pointer to an event in the repository database, while the actual event remains in the WebSphere MQ queue. This saves having to write potentially large events to the repository database.
- Agent side performance:
  WebSphere MQ provides faster performance on the application-specific component side. Using WebSphere MQ, the connector's polling thread picks up an event, places it in the connector's queue, then picks up the next event. This is faster than IDL, which requires the connector's polling thread to pick up an event, go over the network into the server process, store the event persistently in the repository database, then pick up the next event.

JMS

Enables communication between the connector and client connector framework using Java Messaging Service (JMS).

If you select JMS as the delivery transport, additional JMS properties such as jms.MessageBrokerName, jms.FactoryClassName, jms.Password, and jms.UserName, appear in Connector Configurator. The first two of these properties are required for this transport.

Important: There may be a memory limitation if you use the JMS transport mechanism for a connector in the following environment:

- AIX 5.0
- WebSphere MQ 5.3.0.1
- When ICS is the integration broker

In this environment, you may experience difficulty starting both the connector controller (on the server side) and the connector (on the client side) due to memory use within the WebSphere MQ client. If your installation uses less than 768M of process heap size, IBM recommends that you set:

- The LDR_CNTRL environment variable in the CWSharedEnv.sh script.

  This script resides in the \bin directory below the product directory. With a text editor, add the following line as the first line in the CWSharedEnv.sh script:

  ```
  export LDR_CNTRL=MAXDATA=0x30000000
  ```

This line restricts heap memory usage to a maximum of 768 MB (3 segments * 256 MB). If the process memory grows more than this limit, page swapping can occur, which can adversely affect the performance of your system.

- The `IPCCBaseAddress` property to a value of 11 or 12. For more information on this property, see the *System Installation Guide for UNIX*.

DuplicateEventElimination

When you set this property to `true`, a JMS-enabled connector can ensure that duplicate events are not delivered to the delivery queue. To use this feature, the connector must have a unique event identifier set as the business object's **ObjectEventId** attribute in the application-specific code. This is done during connector development.

This property can also be set to `false`.

Note: When `DuplicateEventElimination` is set to `true`, you must also configure the `MonitorQueue` property to enable guaranteed event delivery.

FaultQueue

If the connector experiences an error while processing a message then the connector moves the message to the queue specified in this property, along with a status indicator and a description of the problem.

The default value is `CONNECTORNAME/FAULTQUEUE`.

JvmMaxHeapSize

The maximum heap size for the agent (in megabytes). This property is applicable only if the `RepositoryDirectory` value is <REMOTE>.

The default value is 128m.

JvmMaxNativeStackSize

The maximum native stack size for the agent (in kilobytes). This property is applicable only if the `RepositoryDirectory` value is <REMOTE>.

The default value is 128k.

JvmMinHeapSize

The minimum heap size for the agent (in megabytes). This property is applicable only if the `RepositoryDirectory` value is <REMOTE>.

The default value is 1m.

jms.FactoryClassName

Specifies the class name to instantiate for a JMS provider. You *must* set this connector property when you choose `JMS` as your delivery transport mechanism (DeliveryTransport).

The default is `CxCommon.Messaging.jms.IBMMQSeriesFactory`.

jms.MessageBrokerName

Specifies the broker name to use for the JMS provider. You *must* set this connector property when you choose `JMS` as your delivery transport mechanism (DeliveryTransport).

The default is `crossworlds.queue.manager`.

jms.NumConcurrentRequests

Specifies the maximum number of concurrent service call requests that can be sent to a connector at the same time. Once that maximum is reached, new service calls block and wait for another request to complete before proceeding.

The default value is 10.

jms.Password

Specifies the password for the JMS provider. A value for this property is optional.

There is no default.

jms.UserName

Specifies the user name for the JMS provider. A value for this property is optional.

There is no default.

ListenerConcurrency

This property supports multi-threading in MQ Listener when ICS is the integration broker. It enables batch writing of multiple events to the database, thus improving system performance. The default value is 1.

This property applies only to connectors using MQ transport. The `DeliveryTransport` property must be set to `MQ`.

Locale

Specifies the language code, country or territory, and, optionally, the associated character code set. The value of this property determines such cultural conventions as collation and sort order of data, date and time formats, and the symbols used in monetary specifications.

A locale name has the following format:

ll_TT.codeset

where:

ll	a two-character language code (usually in lower case)
TT	a two-letter country or territory code (usually in upper case)
codeset	the name of the associated character code set; this portion of the name is often optional.

By default, only a subset of supported locales appears in the drop list. To add other supported values to the drop list, you must manually modify the

\Data\Std\stdConnProps.xml file in the product directory. For more information, see the appendix on Connector Configurator.

The default value is en_US. If the connector has not been globalized, the only valid value for this property is en_US. To determine whether a specific connector has been globalized, see the connector version list on these websites:

http://www.ibm.com/software/websphere/wbiadapters/infocenter, or
http://www.ibm.com/websphere/integration/wicserver/infocenter

LogAtInterchangeEnd

Applicable only if RespositoryDirectory is <REMOTE>.

Specifies whether to log errors to the integration broker's log destination. Logging to the broker's log destination also turns on e-mail notification, which generates e-mail messages for the MESSAGE_RECIPIENT specified in the InterchangeSystem.cfg file when errors or fatal errors occur.

For example, when a connector loses its connection to its application, if LogAtInterChangeEnd is set to true, an e-mail message is sent to the specified message recipient. The default is false.

MaxEventCapacity

The maximum number of events in the controller buffer. This property is used by flow control and is applicable only if the value of the RepositoryDirectory property is <REMOTE>.

The value can be a positive integer between 1 and 2147483647. The default value is 2147483647.

MessageFileName

The name of the connector message file. The standard location for the message file is \connectors\messages. Specify the message filename in an absolute path if the message file is not located in the standard location.

If a connector message file does not exist, the connector uses InterchangeSystem.txt as the message file. This file is located in the product directory.

Note: To determine whether a specific connector has its own message file, see the individual adapter user guide.

MonitorQueue

The logical queue that the connector uses to monitor duplicate events. It is used only if the DeliveryTransport property value is JMS and DuplicateEventElimination is set to TRUE.

The default value is CONNECTORNAME/MONITORQUEUE

OADAutoRestartAgent

Valid only when the RepositoryDirectory is <REMOTE>.

Specifies whether the connector uses the automatic and remote restart feature. This feature uses the MQ-triggered Object Activation Daemon (OAD) to restart the connector after an abnormal shutdown, or to start a remote connector from System Monitor.

This property must be set to `true`to enable the automatic and remote restart feature. For information on how to configure the MQ-triggered OAD feature. see the *Installation Guide for Windows* or *for UNIX*.

The default value is `false`.

OADMaxNumRetry

Valid only when the `RepositoryDirectory` is <REMOTE>.

Specifies the maximum number of times that the MQ-triggered OAD automatically attempts to restart the connector after an abnormal shutdown. The `OADAutoRestartAgent` property must be set to `true` for this property to take effect.

The default value is `1000`.

OADRetryTimeInterval

Valid only when the `RepositoryDirectory` is <REMOTE>.

Specifies the number of minutes in the retry-time interval for the MQ-triggered OAD. If the connector agent does not restart within this retry-time interval, the connector controller asks the OAD to restart the connector agent again. The OAD repeats this retry process as many times as specified by the `OADMaxNumRetry` property. The `OADAutoRestartAgent` property must be set to `true` for this property to take effect.

The default is `10`.

PollEndTime

Time to stop polling the event queue. The format is `HH:MM`, where *HH* represents 0-23 hours, and *MM* represents 0-59 seconds.

You must provide a valid value for this property. The default value is `HH:MM`, but must be changed.

PollFrequency

The amount of time between polling actions. Set `PollFrequency` to one of the following values:

- The number of milliseconds between polling actions.
- The word `key`, which causes the connector to poll only when you type the letter `p` in the connector's Command Prompt window. Enter the word in lowercase.
- The word `no`, which causes the connector not to poll. Enter the word in lowercase.

The default is `10000`.

Important: Some connectors have restrictions on the use of this property. To determine whether a specific connector does, see the installing and configuring chapter of its adapter guide.

PollQuantity

Designates the number of items from the application that the connector should poll for. If the adapter has a connector-specific property for setting the poll quantity, the value set in the connector-specific property will override the standard property value.

PollStartTime

The time to start polling the event queue. The format is *HH:MM*, where *HH* represents 0-23 hours, and *MM* represents 0-59 seconds.

You must provide a valid value for this property. The default value is `HH:MM`, but must be changed.

RequestQueue

The queue that is used by the integration broker to send business objects to the connector.

The default value is `CONNECTOR/REQUESTQUEUE`.

RepositoryDirectory

The location of the repository from which the connector reads the XML schema documents that store the meta-data for business object definitions.

When the integration broker is ICS, this value must be set to <REMOTE> because the connector obtains this information from the InterChange Server repository.

When the integration broker is a WebSphere message broker or WAS, this value must be set to <*local directory*>.

ResponseQueue

Applicable only if `DeliveryTransport` is JMS and required only if `RepositoryDirectory` is <REMOTE>.

Designates the JMS response queue, which delivers a response message from the connector framework to the integration broker. When the integration broker is ICS, the server sends the request and waits for a response message in the JMS response queue.

RestartRetryCount

Specifies the number of times the connector attempts to restart itself. When used for a parallel connector, specifies the number of times the master connector application-specific component attempts to restart the slave connector application-specific component.

The default is `3`.

RestartRetryInterval

Specifies the interval in minutes at which the connector attempts to restart itself. When used for a parallel connector, specifies the interval at which the master connector application-specific component attempts to restart the slave connector application-specific component. Possible values ranges from 1 to 2147483647.

The default is 1.

RHF2MessageDomain

WebSphere MQ Integrator broker only.

This property allows you to configure the value of the field domain name in the JMS header. When data is sent to WMQI over JMS transport, the adapter framework writes JMS header information, with a domain name and a fixed value of `mrm`. A connfigurable domain name enables users to track how the WMQI broker processes the message data.

A sample header would look like this:

```
<mcd><Msd>mrm</Msd><Set>3</Set><Type>
Retek_POPhyDesc</Type><Fmt>CwXML</Fmt></mcd>
```

The default value is `mrm`, but it may also be set to `xml`. This property only appears when DeliveryTransport is set to `JMS`and WireFormat is set to `CwXML`.

SourceQueue

Applicable only if `DeliveryTransport` is JMS and ContainerManagedEvents is specified.

Designates the JMS source queue for the connector framework in support of guaranteed event delivery for JMS-enabled connectors that use a JMS event store. For further information, see "ContainerManagedEvents" on page 485.

The default value is `CONNECTOR/SOURCEQUEUE`.

SynchronousRequestQueue

Applicable only if `DeliveryTransport` is JMS.

Delivers request messages that require a synchronous response from the connector framework to the broker. This queue is necessary only if the connector uses synchronous execution. With synchronous execution, the connector framework sends a message to the `SynchronousRequestQueue` and waits for a response back from the broker on the `SynchronousResponseQueue`. The response message sent to the connector bears a correlation ID that matches the ID of the original message.

The default is `CONNECTORNAME/SYNCHRONOUSREQUESTQUEUE`

SynchronousResponseQueue

Applicable only if `DeliveryTransport` is JMS.

Delivers response messages sent in reply to a synchronous request from the broker to the connector framework. This queue is necessary only if the connector uses synchronous execution.

The default is `CONNECTORNAME/SYNCHRONOUSRESPONSEQUEUE`

SynchronousRequestTimeout

Applicable only if `DeliveryTransport` is JMS.

Specifies the time in minutes that the connector waits for a response to a synchronous request. If the response is not received within the specified time, then the connector moves the original synchronous request message into the fault queue along with an error message.

The default value is 0.

WireFormat

Message format on the transport.

- If the `RepositoryDirectory` is a local directory, the setting is `CwXML`.
- If the value of `RepositoryDirectory` is <REMOTE>, the setting is`CwBO`.

WsifSynchronousRequest Timeout

WAS integration broker only.

Specifies the time in minutes that the connector waits for a response to a synchronous request. If the response is not received within the specified, time then the connector moves the original synchronous request message into the fault queue along with an error message.

The default value is 0.

XMLNameSpaceFormat

WebSphere message brokers and WAS integration broker only.

A strong property that allows the user to specify short and long name spaces in the XML format of business object definitions.

The default value is `short`.

Appendix B. Connector Configurator

This appendix describes how to use Connector Configurator to set configuration property values for your adapter.

You use Connector Configurator to:

- Create a connector-specific property template for configuring your connector
- Create a configuration file
- Set properties in a configuration file

Note:

In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes and follow the conventions for each operating system.

The topics covered in this appendix are:

Overview of Connector Configurator

Connector Configurator allows you to configure the connector component of your adapter for use with these integration brokers:

- WebSphere InterChange Server (ICS)
- WebSphere MQ Integrator, WebSphere MQ Integrator Broker, and WebSphere Business Integration Message Broker, collectively referred to as the WebSphere Message Brokers (WMQI)
- WebSphere Application Server (WAS)

You use Connector Configurator to:

- Create a **connector-specific property template** for configuring your connector.
- Create a **connector configuration file**; you must create one configuration file for each connector you install.
- Set properties in a configuration file.
  You may need to modify the default values that are set for properties in the connector templates. You must also designate supported business object definitions and, with ICS, maps for use with collaborations as well as specify messaging, logging and tracing, and data handler parameters, as required.

The mode in which you run Connector Configurator, and the configuration file type you use, may differ according to which integration broker you are running. For example, if WMQI is your broker, you run Connector Configurator directly, and not from within System Manager (see "Running Configurator in stand-alone mode" on page 496).

Connector configuration properties include both standard configuration properties (the properties that all connectors have) and connector-specific properties (properties that are needed by the connector for a specific application or technology).

Because **standard properties** are used by all connectors, you do not need to define those properties from scratch; Connector Configurator incorporates them into your configuration file as soon as you create the file. However, you do need to set the value of each standard property in Connector Configurator.

The range of standard properties may not be the same for all brokers and all configurations. Some properties are available only if other properties are given a specific value. The Standard Properties window in Connector Configurator will show the properties available for your particular configuration.

For **connector-specific properties**, however, you need first to define the properties and then set their values. You do this by creating a connector-specific property template for your particular adapter. There may already be a template set up in your system, in which case, you simply use that. If not, follow the steps in "Creating a new template" on page 497 to set up a new one.

Note: Connector Configurator runs only in a Windows environment. If you are running the connector in a UNIX environment, use Connector Configurator in Windows to modify the configuration file and then copy the file to your UNIX environment.

Starting Connector Configurator

You can start and run Connector Configurator in either of two modes:

- Independently, in stand-alone mode
- From System Manager

Running Configurator in stand-alone mode

You can run Connector Configurator independently and work with connector configuration files, irrespective of your broker.

To do so when the broker is IBM WebSphere InterChange Server:

- From **Start>Programs**, click **IBM WebSphere InterChange Server>IBM WebSphere Business Integration Toolset>Development>Connector Configurator**.
- Select **File>New>Configuration File**.
- When you click the pull-down menu next to **System Connectivity Integration Broker**, you can select ICS connectivity.

To do so when you have WebSphere Business Integration Adapters and another broker installed:

- From **Start>Programs**, click **IBM WebSphere Business Integration Adapters>Tools>Connector Configurator**.
- Select **File>New>Connector Configuration**.
- When you click the pull-down menu next to **System Connectivity Integration Broker:**, you can select WMQI or WAS connectivity, depending on your broker.

You may choose to run Connector Configurator independently to generate the file, and then connect to System Manager to save it in a System Manager project (see "Completing a configuration file" on page 501.)

Running Configurator from System Manager

You can run Connector Configurator from System Manager.

To run Connector Configurator:

1. Open the System Manager.
2. In the System Manager window, expand the **Integration Component Libraries** icon and highlight **Connectors**.
3. From the System Manager menu bar, click **Tools>Connector Configurator**. The Connector Configurator window opens and displays a **New Connector** dialog box.
4. When you click the pull-down menu next to **System Connectivity Integration Broker**, you can select ICS, WebSphere Message Brokers or WAS, depending on your broker.

To edit an existing configuration file:

1. In the System Manager window, select any of the configuration files listed in the Connector folder and right-click on it. Connector Configurator opens and displays the configuration file with the integration broker type and file name at the top.
2. Click the Standard Properties tab to see which properties are included in this configuration file.

Creating a connector-specific property template

To create a configuration file for your connector, you need a connector-specific property template as well as the system-supplied standard properties.

You can create a brand-new template for the connector-specific properties of your connector, or you can use an existing file as the template.

- To create a new template, see "Creating a new template" on page 497.
- To use an existing file, simply modify an existing template and save it under the new name.

Creating a new template

This section describes how you create properties in the template, define general characteristics and values for those properties, and specify any dependencies between the properties. Then you save the template and use it as the base for creating a new connector configuration file.

To create a template:

1. Click **File>New>Connector-Specific Property Template**.
2. The **Connector-Specific Property Template** dialog box appears, with the following fields:
   - **Template**, and **Name**

     Enter a unique name that identifies the connector, or type of connector, for which this template will be used. You will see this name again when you open the dialog box for creating a new configuration file from a template.

- **Old Template**, and **Select the Existing Template to Modify**

  The names of all currently available templates are displayed in the Template Name display.
- To see the connector-specific property definitions in any template, select that template's name in the **Template Name** display. A list of the property definitions contained in that template will appear in the **Template Preview** display. You can use an existing template whose property definitions are similar to those required by your connector as a starting point for your template.

3. Select a template from the **Template Name** display, enter that template name in the **Find Name** field (or highlight your selection in **Template Name**), and click **Next**.

If you do not see any template that displays the connector-specific properties used by your connector, you will need to create one.

Specifying general characteristics

When you click **Next** to select a template, the **Properties - Connector-Specific Property Template** dialog box appears. The dialog box has tabs for General characteristics of the defined properties and for Value restrictions. The General display has the following fields:

- **General:**
  Property Type
  Updated Method
  Description
- **Flags**
  Standard flags
- **Custom Flag**
  Flag

After you have made selections for the general characteristics of the property, click the **Value** tab.

Specifying values

The **Value** tab enables you to set the maximum length, the maximum multiple values, a default value, or a value range for the property. It also allows editable values. To do so:

1. Click the **Value** tab. The display panel for Value replaces the display panel for General.
2. Select the name of the property in the **Edit properties** display.
3. In the fields for **Max Length** and **Max Multiple Values**, make any changes. The changes will not be accepted unless you also open the **Property Value** dialog box for the property, described in the next step.
4. Right-click the box in the top left-hand corner of the value table and click **Add**. A **Property Value** dialog box appears. Depending on the property type, the dialog box allows you to enter either a value, or both a value and range. Enter the appropriate value or range, and click **OK**.
5. The **Value** panel refreshes to display any changes you made in **Max Length** and **Max Multiple Values**. It displays a table with three columns:

   The **Value** column shows the value that you entered in the **Property Value** dialog box, and any previous values that you created.

   The **Default Value** column allows you to designate any of the values as the default.

The **Value Range** shows the range that you entered in the **Property Value** dialog box.

After a value has been created and appears in the grid, it can be edited from within the table display. To make a change in an existing value in the table, select an entire row by clicking on the row number. Then right-click in the **Value** field and click **Edit Value**.

Setting dependencies

When you have made your changes to the **General** and **Value** tabs, click **Next**. The **Dependences - Connector-Specific Property Template** dialog box appears.

A dependent property is a property that is included in the template and used in the configuration file *only if* the value of another property meets a specific condition. For example, `PollQuantity` appears in the template only if JMS is the transport mechanism and `DuplicateEventElimination` is set to `True`.
To designate a property as dependent and to set the condition upon which it depends, do this:

1. In the **Available Properties** display, select the property that will be made dependent.
2. In the **Select Property** field, use the drop-down menu to select the property that will hold the conditional value.
3. In the **Condition Operator** field, select one of the following:

   == (equal to)

   != (not equal to)

   > (greater than)

   < (less than)

   >= (greater than or equal to)

   <=(less than or equal to)
4. In the **Conditional Value** field, enter the value that is required in order for the dependent property to be included in the template.
5. With the dependent property highlighted in the **Available Properties** display, click an arrow to move it to the **Dependent Property** display.
6. Click **Finish**. Connector Configurator stores the information you have entered as an XML document, under `\data\app` in the`\bin` directory where you have installed Connector Configurator.

Creating a new configuration file

When you create a new configuration file, your first step is to select an integration broker. The broker you select determines the properties that will appear in the configuration file.

To select a broker:

- In the Connector Configurator home menu, click **File>New>Connector Configuration**. The **New Connector** dialog box appears.
- In the I**ntegration Broker** field, select ICS, WebSphere Message Brokers or WAS connectivity.
- Complete the remaining fields in the **New Connector** window, as described later in this chapter.

You can also do this:

- In the System Manager window, right-click on the **Connectors** folder and select **Create New Connector**. Connector Configurator opens and displays the **New Connector** dialog box.

Creating a configuration file from a connector-specific template

Once a connector-specific template has been created, you can use it to create a configuration file:

1. Click **File>New>Connector Configuration**.
2. The **New Connector** dialog box appears, with the following fields:
   - **Name**

     Enter the name of the connector. Names are case-sensitive. The name you enter must be unique, and must be consistent with the file name for a connector that is installed on the system.

     Important: Connector Configurator does not check the spelling of the name that you enter. You must ensure that the name is correct.
   - **System Connectivity**

     Click ICS or WebSphere Message Brokers or WAS.
   - **Select Connector-Specific Property Template**

     Type the name of the template that has been designed for your connector. The available templates are shown in the **Template Name** display. When you select a name in the Template Name display, the **Property Template Preview** display shows the connector-specific properties that have been defined in that template.

     Select the template you want to use and click **OK**.
3. A configuration screen appears for the connector that you are configuring. The title bar shows the integration broker and connector names. You can fill in all the field values to complete the definition now, or you can save the file and complete the fields later.
4. To save the file, click **File>Save>To File** or **File>Save>To Project**. To save to a project, System Manager must be running.
   If you save as a file, the **Save File Connector** dialog box appears. Choose `*.cfg` as the file type, verify in the File Name field that the name is spelled correctly and has the correct case, navigate to the directory where you want to locate the file, and click **Save**. The status display in the message panel of Connector Configurator indicates that the configuration file was successfully created.

   Important: The directory path and name that you establish here must match the connector configuration file path and name that you supply in the startup file for the connector.
5. To complete the connector definition, enter values in the fields for each of the tabs of the Connector Configurator window, as described later in this chapter.

Using an existing file

You may have an existing file available in one or more of the following formats:

- A connector definition file.
  This is a text file that lists properties and applicable default values for a specific connector. Some connectors include such a file in a `\repository` directory in their delivery package (the file typically has the extension `.txt`; for example, CN_XML.txt for the XML connector).

- An ICS repository file.
  Definitions used in a previous ICS implementation of the connector may be available to you in a repository file that was used in the configuration of that connector. Such a file typically has the extension `.in` or `.out`.
- A previous configuration file for the connector.
  Such a file typically has the extension *.cfg.

Although any of these file sources may contain most or all of the connector-specific properties for your connector, the connector configuration file will not be complete until you have opened the file and set properties, as described later in this chapter.

To use an existing file to configure a connector, you must open the file in Connector Configurator, revise the configuration, and then resave the file.

Follow these steps to open a *.txt, *.cfg, or *.in file from a directory:

1. In Connector Configurator, click **File>Open>From File**.
2. In the **Open File Connector** dialog box, select one of the following file types to see the available files:
   - Configuration (`*.cfg`)
   - ICS Repository (`*.in`, `*.out`)

     Choose this option if a repository file was used to configure the connector in an ICS environment. A repository file may include multiple connector definitions, all of which will appear when you open the file.
   - All files (*.*)

     Choose this option if a `*.txt` file was delivered in the adapter package for the connector, or if a definition file is available under another extension.
3. In the directory display, navigate to the appropriate connector definition file, select it, and click **Open**.

Follow these steps to open a connector configuration from a System Manager project:

1. Start System Manager. A configuration can be opened from or saved to System Manager only if System Manager has been started.
2. Start Connector Configurator.
3. Click **File>Open>From Project**.

Completing a configuration file

When you open a configuration file or a connector from a project, the Connector Configurator window displays the configuration screen, with the current attributes and values.

The title of the configuration screen displays the integration broker and connector name as specified in the file. Make sure you have the correct broker. If not, change the broker value before you configure the connector. To do so:

1. Under the **Standard Properties** tab, select the value field for the `BrokerType` property. In the drop-down menu, select the value `ICS`, `WMQI`, or `WAS`.
2. The Standard Properties tab will display the properties associated with the selected broker. You can save the file now or complete the remaining configuration fields, as described in "Specifying supported business object definitions" on page 504..

3. When you have finished your configuration, click **File>Save>To Project** or **File>Save>To File**.

   If you are saving to file, select `*.cfg` as the extension, select the correct location for the file and click **Save**.

   If multiple connector configurations are open, click **Save All to File** to save all of the configurations to file, or click **Save All to Project** to save all connector configurations to a System Manager project.

   Before it saves the file, Connector Configurator checks that values have been set for all required standard properties. If a required standard property is missing a value, Connector Configurator displays a message that the validation failed. You must supply a value for the property in order to save the configuration file.

Setting the configuration file properties

When you create and name a new connector configuration file, or when you open an existing connector configuration file, Connector Configurator displays a configuration screen with tabs for the categories of required configuration values.

Connector Configurator requires values for properties in these categories for connectors running on all brokers:

- Standard Properties
- Connector-specific Properties
- Supported Business Objects
- Trace/Log File values
- Data Handler (applicable for connectors that use JMS messaging with guaranteed event delivery)

Note: For connectors that use JMS messaging, an additional category may display, for configuration of data handlers that convert the data to business objects.

For connectors running on **ICS**, values for these properties are also required:

- Associated Maps
- Resources
- Messaging (where applicable)

Important: Connector Configurator accepts property values in either English or non-English character sets. However, the names of both standard and connector-specific properties, and the names of supported business objects, must use the English character set only.

Standard properties differ from connector-specific properties as follows:

- Standard properties of a connector are shared by both the application-specific component of a connector and its broker component. All connectors have the same set of standard properties. These properties are described in Appendix A of each adapter guide. You can change some but not all of these values.
- Application-specific properties apply only to the application-specific component of a connector, that is, the component that interacts directly with the application. Each connector has application-specific properties that are unique to its application. Some of these properties provide default values and some do not; you can modify some of the default values. The installation and configuration chapters of each adapter guide describe the application-specific properties and the recommended values.

The fields for **Standard Properties** and **Connector-Specific Properties** are color-coded to show which are configurable:

- A field with a grey background indicates a standard property. You can change the value but cannot change the name or remove the property.
- A field with a white background indicates an application-specific property. These properties vary according to the specific needs of the application or connector. You can change the value and delete these properties.
- Value fields are configurable.
- The **Update Method** field is informational and not configurable. This field specifies the action required to activate a property whose value has changed.

Setting standard connector properties

To change the value of a standard property:

1. Click in the field whose value you want to set.
2. Either enter a value, or select one from the drop-down menu if it appears.
3. After entering all the values for the standard properties, you can do one of the following:
   - To discard the changes, preserve the original values, and exit Connector Configurator, click **File>Exit** (or close the window), and click **No** when prompted to save changes.
   - To enter values for other categories in Connector Configurator, select the tab for the category. The values you enter for **Standard Properties** (or any other category) are retained when you move to the next category. When you close the window, you are prompted to either save or discard the values that you entered in all the categories as a whole.
   - To save the revised values, click **File>Exit** (or close the window) and click **Yes** when prompted to save changes. Alternatively, click **Save>To File** from either the File menu or the toolbar.

Setting application-specific configuration properties

For application-specific configuration properties, you can add or change property names, configure values, delete a property, and encrypt a property. The default property length is 255 characters.

1. Right-click in the top left portion of the grid. A pop-up menu bar will appear. Click **Add** to add a property. To add a child property, right-click on the parent row number and click **Add child**.
2. Enter a value for the property or child property.
3. To encrypt a property, select the **Encrypt** box.
4. Choose to save or discard changes, as described for "Setting standard connector properties."

The Update Method displayed for each property indicates whether a component or agent restart is necessary to activate changed values.

Important: Changing a preset application-specific connector property name may cause a connector to fail. Certain property names may be needed by the connector to connect to an application or to run properly.

Encryption for connector properties

Application-specific properties can be encrypted by selecting the **Encrypt** check box in the **Edit Property** window. To decrypt a value, click to clear the **Encrypt**

check box, enter the correct value in the **Verification** dialog box, and click **OK**. If the entered value is correct, the value is decrypted and displays.

The adapter user guide for each connector contains a list and description of each property and its default value.

If a property has multiple values, the **Encrypt** check box will appear for the first value of the property. When you select **Encrypt**, all values of the property will be encrypted. To decrypt multiple values of a property, click to clear the **Encrypt** check box for the first value of the property, and then enter the new value in the **Verification** dialog box. If the input value is a match, all multiple values will decrypt.

Update method

Refer to the descriptions of update methods found in the *Standard configuration properties for connectors* appendix, under "Setting and updating property values" on page 480.

Specifying supported business object definitions

Use the **Supported Business Objects** tab in Connector Configurator to specify the business objects that the connector will use. You must specify both generic business objects and application-specific business objects, and you must specify associations for the maps between the business objects.

Note: Some connectors require that certain business objects be specified as supported in order to perform event notification or additional configuration (using meta-objects) with their applications. For more information, see the *Connector Development Guide for C++* or the *Connector Development Guide for Java*.

If ICS is your broker

To specify that a business object definition is supported by the connector, or to change the support settings for an existing business object definition, click the **Supported Business Objects** tab and use the following fields.

Business object name: To designate that a business object definition is supported by the connector, with System Manager running:

1. Click an empty field in the **Business Object Name** list. A drop-down list displays, showing all the business object definitions that exist in the System Manager project.
2. Click on a business object to add it.
3. Set the **Agent Support** (described below) for the business object.
4. In the File menu of the Connector Configurator window, click **Save to Project**. The revised connector definition, including designated support for the added business object definition, is saved to the project in System Manager.

To delete a business object from the supported list:

1. To select a business object field, click the number to the left of the business object.
2. From the **Edit** menu of the Connector Configurator window, click **Delete Row**. The business object is removed from the list display.
3. From the **File** menu, click **Save to Project**.

Deleting a business object from the supported list changes the connector definition and makes the deleted business object unavailable for use in this implementation of this connector. It does not affect the connector code, nor does it remove the business object definition itself from System Manager.

Agent support: If a business object has Agent Support, the system will attempt to use that business object for delivering data to an application via the connector agent.

Typically, application-specific business objects for a connector are supported by that connector's agent, but generic business objects are not.

To indicate that the business object is supported by the connector agent, check the **Agent Support** box. The Connector Configurator window does not validate your Agent Support selections.

Maximum transaction level: The maximum transaction level for a connector is the highest transaction level that the connector supports.

For most connectors, `Best Effort` is the only possible choice.

You must restart the server for changes in transaction level to take effect.

If a WebSphere Message Broker is your broker

If you are working in stand-alone mode (not connected to System Manager), you must enter the business name manually.

If you have System Manager running, you can select the empty box under the **Business Object Name** column in the **Supported Business Objects** tab. A combo box appears with a list of the business object available from the Integration Component Library project to which the connector belongs. Select the business object you want from the list.

The **Message Set ID** is an optional field for WebSphere Business Integration Message Broker 5.0, and need not be unique if supplied. However, for WebSphere MQ Integrator and Integrator Broker 2.1, you must supply a unique **ID**.

If WAS is your broker

When WebSphere Application Server is selected as your broker type, Connector Configurator does not require message set IDs. The **Supported Business Objects** tab shows a **Business Object Name** column only for supported business objects.

If you are working in stand-alone mode (not connected to System Manager), you must enter the business object name manually.

If you have System Manager running, you can select the empty box under the Business Object Name column in the Supported Business Objects tab. A combo box appears with a list of the business objects available from the Integration Component Library project to which the connector belongs. Select the business object you want from this list.

Associated maps (ICS only)

Each connector supports a list of business object definitions and their associated maps that are currently active in WebSphere InterChange Server. This list appears when you select the **Associated Maps** tab.

The list of business objects contains the application-specific business object which the agent supports and the corresponding generic object that the controller sends to the subscribing collaboration. The association of a map determines which map will be used to transform the application-specific business object to the generic business object or the generic business object to the application-specific business object.

If you are using maps that are uniquely defined for specific source and destination business objects, the maps will already be associated with their appropriate business objects when you open the display, and you will not need (or be able) to change them.

If more than one map is available for use by a supported business object, you will need to explicitly bind the business object with the map that it should use.

The **Associated Maps** tab displays the following fields:

- **Business Object Name**

  These are the business objects supported by this connector, as designated in the **Supported Business Objects** tab. If you designate additional business objects under the Supported Business Objects tab, they will be reflected in this list after you save the changes by choosing **Save to Project** from the **File** menu of the Connector Configurator window.
- **Associated Maps**

  The display shows all the maps that have been installed to the system for use with the supported business objects of the connector. The source business object for each map is shown to the left of the map name, in the **Business Object Name** display.
- **Explicit**

  In some cases, you may need to explicitly bind an associated map.

  Explicit binding is required only when more than one map exists for a particular supported business object. When ICS boots, it tries to automatically bind a map to each supported business object for each connector. If more than one map takes as its input the same business object, the server attempts to locate and bind one map that is the superset of the others.

  If there is no map that is the superset of the others, the server will not be able to bind the business object to a single map, and you will need to set the binding explicitly.

  To explicitly bind a map:

  1. In the **Explicit** column, place a check in the check box for the map you want to bind.
  2. Select the map that you intend to associate with the business object.
  3. In the **File** menu of the Connector Configurator window, click **Save to Project**.
  4. Deploy the project to ICS.
  5. Reboot the server for the changes to take effect.

Resources (ICS)

The **Resource** tab allows you to set a value that determines whether and to what extent the connector agent will handle multiple processes concurrently, using connector agent parallelism.
Not all connectors support this feature. If you are running a connector agent that

was designed in Java to be multi-threaded, you are advised not to use this feature, since it is usually more efficient to use multiple threads than multiple processes.

Messaging (ICS)

The messaging properties are available only if you have set MQ as the value of the `DeliveryTransport` standard property and ICS as the broker type. These properties affect how your connector will use queues.

Setting trace/log file values

When you open a connector configuration file or a connector definition file, Connector Configurator uses the logging and tracing values of that file as default values. You can change those values in Connector Configurator.

To change the logging and tracing values:

1. Click the **Trace/Log Files** tab.
2. For either logging or tracing, you can choose to write messages to one or both of the following:
   - To console (STDOUT):
     Writes logging or tracing messages to the STDOUT display.

     Note: You can only use the STDOUT option from the **Trace/Log Files** tab for connectors running on the Windows platform.
   - To File:
     Writes logging or tracing messages to a file that you specify. To specify the file, click the directory button (ellipsis), navigate to the preferred location, provide a file name, and click **Save**. Logging or tracing message are written to the file and location that you specify.

     Note: Both logging and tracing files are simple text files. You can use the file extension that you prefer when you set their file names. For tracing files, however, it is advisable to use the extension `.trace` rather than `.trc`, to avoid confusion with other files that might reside on the system. For logging files, `.log` and `.txt` are typical file extensions.

Data handlers

The data handlers section is available for configuration only if you have designated a value of JMS for DeliveryTransport and a value of JMS for ContainerManagedEvents. Not all adapters make use of data handlers.

See the descriptions under `ContainerManagedEvents` in Appendix A, Standard Properties, for values to use for these properties. For additional details, see the *Connector Development Guide for C++* or the *Connector Development Guide for Java.*

Saving your configuration file

When you have finished configuring your connector, save the connector configuration file. Connector Configurator saves the file in the broker mode that you selected during configuration. The title bar of Connector Configurator always displays the broker mode (ICS, WMQI or WAS) that it is currently using.

The file is saved as an XML document. You can save the XML document in three ways:

- From System Manager, as a file with a `*.con` extension in an Integration Component Library, or
- In a directory that you specify.
- In stand-alone mode, as a file with a `*.cfg` extension in a directory folder.

For details about using projects in System Manager, and for further information about deployment, see the following implementation guides:

- For ICS: *Implementation Guide for WebSphere InterChange Server*
- For WebSphere Message Brokers: *Implementing Adapters with WebSphere Message Brokers*
- For WAS: *Implementing Adapters with WebSphere Application Server*

Changing a configuration file

You can change the integration broker setting for an existing configuration file. This enables you to use the file as a template for creating a new configuration file, which can be used with a different broker.

Note: You will need to change other configuration properties as well as the broker mode property if you switch integration brokers.

To change your broker selection within an existing configuration file (optional):

- Open the existing configuration file in Connector Configurator.
- Select the **Standard Properties** tab.
- In the **BrokerType** field of the Standard Properties tab, select the value that is appropriate for your broker.
  When you change the current value, the available tabs and field selections on the properties screen will immediately change, to show only those tabs and fields that pertain to the new broker you have selected.

Completing the configuration

After you have created a configuration file for a connector and modified it, make sure that the connector can locate the configuration file when the connector starts up.

To do so, open the startup file used for the connector, and verify that the location and file name used for the connector configuration file match exactly the name you have given the file and the directory or path where you have placed it.

Using Connector Configurator in a globalized environment

Connector Configurator is globalized and can handle character conversion between the configuration file and the integration broker. Connector Configurator uses native encoding. When it writes to the configuration file, it uses UTF-8 encoding.

Connector Configurator supports non-English characters in:

- All value fields
- Log file and trace file path (specified in the **Trace/Log files** tab)

The drop list for the `CharacterEncoding` and `Locale` standard configuration properties displays only a subset of supported values. To add other values to the drop list, you must manually modify the `\Data\Std\stdConnProps.xml` file in the product directory.

For example, to add the locale en_GB to the list of values for the Locale property, open the stdConnProps.xml file and add the line in boldface type below:

```
<Property name="Locale"
isRequired="true"
updateMethod="component restart">
                <ValidType>String</ValidType>
            <ValidValues>
                                <Value>ja_JP</Value>
                                <Value>ko_KR</Value>
                                <Value>zh_CN</Value>
                                <Value>zh_TW</Value>
                                <Value>fr_FR</Value>
                                <Value>de_DE</Value>
                                <Value>it_IT</Value>
                                <Value>es_ES</Value>
                                <Value>pt_BR</Value>
                                <Value>en_US</Value>
                                <Value>en_GB</Value>
                    <DefaultValue>en_US</DefaultValue>
            </ValidValues>
    </Property>
```

Appendix C. Connector Script Generator

The Connector Script Generator utility creates or modifies the connector script for connectors running on the UNIX platform. Use this tool to do either of the following:

- To generate a new connector startup script for a connector you have added without using the WebSphere Business Integration Adapters installer.
- To modify an existing startup script for a connector to include the correct configuration file path.

To run the Connector Script Generator, do the following:

1. Navigate to the *ProductDir*/bin directory.
2. Enter the command `./ConnConfig.sh`.

   The Connector Script Generator screen appears as shown in Figure 78.

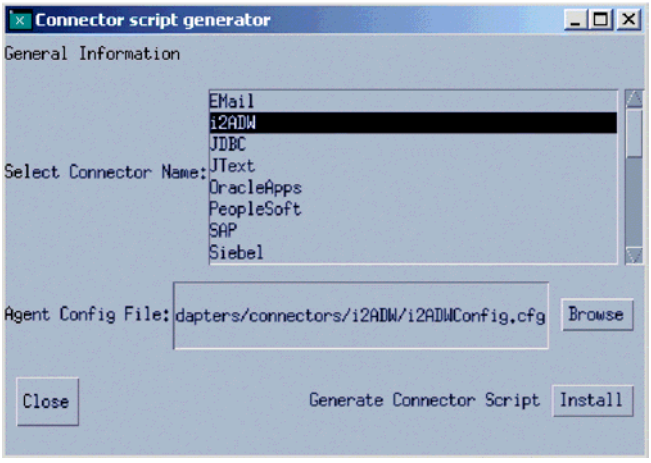

Figure 78. Connector Script Generator.

3. From the Select Connector Name list, select the connector for which the startup script is to be generated.
4. For Agent Config File, specify the connector's configuration file by entering its full-path name or by clicking **Browse** to select a file.
5. To generate or update the connector script, click **Install**.

   The connector_manager_*ConnectorName* file (where *ConnectorName* is the name of the connector you are configuring) is created in the *ProductDir*/bin directory.
6. Click **Close**.

Appendix D. Connector feature checklist

This appendix describes the connector feature checklist.

Guidelines for using the connector feature checklist

The connector feature checklist briefly describes each of the standard features for connectors. The feature list establishes a baseline for the behavior of a connector. Therefore, as you design a new connector, you can use the list as a quick reference to standard connector features.

During the implementation phase for your connector, you can use the feature list to create a specification describing the functionality of your connector. To use the list:

- Check *Full* for each feature that your connector supports.
- Check *Partial* for each feature that your connector partially supports and include notes describing the implementation.
- Check *No* for each feature that the connector does not support.
- Check *N/A* for each feature that is not relevant for the connector. For example, if your connector does not implement event notification, check N/A for all event notification features.

If a feature is not supported according to the standard behavior, check *Partial* and provide additional information.

Standard behavior for request processing

Table 162 lists the standard features for connector handling of business object requests. The table includes a brief description of each feature and a page number of the section in the book containing more information on the feature.

Table 162. Standard features for request processing

Category and name	Description	Supported?
Business Object and Attribute Naming		
Business object names	Business object names should have no semantic value to the connector. Two business objects with the same structure, data, and application-specific information but with different names should process identically in the connector.	__ Full __ Partial __ No __ N/A
Attribute names	Attribute names in a business object should have no semantic value to the connector. Values such as application table name or column name should be stored in the application-specific information field of the attribute and not in the attribute name.	__ Full __ Partial __ No __ N/A
Create		
Create Verb	The connector creates the object in the destination application. The application object includes all values in the business object, including child objects. See “Handling the Create verb” on page 86.	__ Full __ Partial __ No __ N/A
Delete		
Delete Verb	The connector supports the Delete verb, and when processing this verb, it does a true physical delete, not a logical delete. See “Handling the Delete verb” on page 104.	__ Full __ Partial __ No __ N/A

Table 162. Standard features for request processing (continued)

Category and name	Description	Supported?
Logical delete	The connector supports logical deletes operations via the Update verb only. The Delete verb is used only for physical deletes. See "Implications of business objects representing logical Delete events" on page 101.	__ Full __ Partial __ No __ N/A
Exist		
Exist Verb	The connector checks for the existence of an entity in the application database. It returns `SUCCEED` if the object passed in exists in the application database, and `FAIL` if the object does not exist in the application database. See "Handling the Exists verb" on page 105.	__ Full __ Partial __ No __ N/A
Retrieve		
Retrieve Verb	The entire hierarchical image (including all child business objects) is retrieved from application when the Retrieve verb is processed. The retrieve is based only on the key values of the business object. See "Handling the Retrieve verb" on page 89.	__ Full __ Partial __ No __ N/A
Ignore missing child object	If `IgnoreMissingChildObject` is set to true in the business object level application-specific information, the connector returns `SUCCEED` even if not all the children specified in the business object are found in the application. See "Retrieving child objects" on page 92.	__ Full __ Partial __ No __ N/A
RetrieveByContent		
RetrieveBy Content Verb	The entire hierarchical image (including all child objects), based solely on a subset of non-key values, is retrieved. See "Handling the RetrieveByContent verb" on page 95.	__ Full __ Partial __ No __ N/A
Multiple results	If more than one object is retrieved from the application, RetrieveByContent should return the first object and use the return code `MULTIPLE_HITS`. See "Handling the RetrieveByContent verb" on page 95.	__ Full __ Partial __ No __ N/A
Ignore missing child object	If `IgnoreMissingChildObject` is set to true in the business object level application-specific information, the connector returns `SUCCEED` even if not all the children specified in the business object are found in the application.	__ Full __ Partial __ No __ N/A
Update		
After-image support	The connector performs all the steps necessary to make the object in the destination application exactly match the business object received in the `doVerbFor()`call. See "Handling the Update verb" on page 97.	__ Full __ Partial __ No __ N/A
Delta support	Connector processes exactly the objects and verbs that are received in the source business object. The destination application object is updated only by processing the contents of source business object, not by making the application representation match the source business object. [Not currently an IBM standard.]	__ Full __ Partial __ No __ N/A
KeepRelations	When KeepRelations is specified, child relations are not destroyed in the target application. Otherwise, all the child relations are destroyed first, then the child objects sent in from InterChange Server are created and the relations restored. "Destroyed" means a logical or physical delete of the relation to the child, or, in some cases, deletion of the child itself, depending on the functionality of the connector and application. KeepRelations is set as application-specific information on the child array in the parent object (not as text on the child itself). The syntax should be `keeprelations=true`.	__ Full __ Partial __ No __ N/A
Verb Support		

Table 162. Standard features for request processing (continued)

Category and name	Description	Supported?
Subverb support	The connector supports processing of verbs on child objects independent of the verb on the parent object. When a verb is set in a child business object, the connector performs the operation that the child verb indicates, regardless of the verb on the top-level business object. If a verb in a child business object request is not set, the connector can either leave the child verb as `NULL`, set the child verb to the verb in the top-level business object, or set the value of the verb to the operation that the connector needs to perform. See "Verb stability" on page 84.	__ Full __ Partial __ No __ N/A
Verb Stability	Verbs in a business object should remain stable throughout the request and response cycle. When a connector receives an business object request, the hierarchical object returned to InterChange Server should have the same verb(s) as the original request, with the exception of verbs that are set on child business objects that were null in the original request	__ Full __ Partial __ No __ N/A

Standard behavior for the event notification

Table 163 lists standard features for event retrieval and notification.

Table 163. Standard features for event notification

Category and name	Description	Supported?
Connector Properties		
Event distribution	The event retrieval mechanism includes a filter that processes only events that are associated with the connector making the poll call. This feature requires adding a ConnectorId field to the event table so that multiple connectors can use the same event table. Each connector also requires a ConnectorId connector property. This property sets the identifier for a particular instance of a connector and allows the connector to pick up only the events assigned to it. See "Event distribution" on page 131.	__ Full __ Partial __ No __ N/A
PollQuantity	The connector uses the `PollQuantity` connector property to specify the maximum number of events the connector will process for each poll call. If possible, the connector should limit the number of rows retrieved in the poll call to PollQuantity. (For example, in SQL Server, use the `set rowcount` option.) See "Retrieving event records" on page 182.	__ Full __ Partial __ No __ N/A
Event Table		
Event status values	Where applicable, the values are used for event status are described in Table 47 on page 118.	__ Full __ Partial __ No __ N/A
Object key	The object key column must use name-value pairs to set data in a new business object. For example, if ContractId is the name of an attribute in the business object, the object key is: ContractId=45381. The connector should support multiple name-value pairs separated by a delimiter. The delimiter is configurable (PollAttributeDelimiter) and should default to a colon (:). See "Object key" on page 117.	__ Full __ Partial __ No __ N/A
Object name	The object name field should be set to the exact business object name. See "Standard contents of an event record" on page 116.	__ Full __ Partial __ No __ N/A

Table 163. Standard features for event notification (continued)

Category and name	Description	Supported?
Priority	Priority is 0-n, with 0 being the highest priority. The connector polls and processes events in order of priority. Note that no decrementing is done, which could, in rare circumstances, lead to low priority events being shut out (not processed). See "Processing events by event priority" on page 131.	__ Full __ Partial __ No __ N/A
Miscellaneous Features		
Archiving	An event is archived once the connector has processed it, whether or not the event was successfully delivered to InterChange Server. The event status is kept in the archive table and is one of the following: • Success. The event was detected, and an object was created and sent to InterChange Server. • Unsubscribed. The event was detected, but the connector did not have a subscription for that event/verb combination, so the event was not sent to InterChange Server. • Error. The event was detected, but the connector encountered an error in trying to process the event, either in the process of building a business object or in posting the object to InterChange Server.	__ Full __ Partial __ No __ N/A
CDK method `gotApplEvent()`	The connector should call `gotApplEvent()`only from within `pollForEvents()`.	__ Full __ Partial __ No __ N/A
Delta event notification	An event can be created that represents only the changes to a hierarchical business object, such as the addition or deletion of order lines, without creating an update event for the entire business object. [Not currently an IBM Standard]	__ Full __ Partial __ No __ N/A
Future event processing	The mechanism for specifying a future date or time at which an event should be processed. The connector does not process the event until that date or time. [Not currently an IBM Standard]	__ Full __ Partial __ No __ N/A
In-Progress event recovery	When restarted, a connector checks the event table for events that have a status of `IN_PROGRESS`. If any exist, the connector does one of the following: • `PropValue = FailOnStartup`: Logs a fatal error and sends an email notification. • `PropValue = Reprocess`: Submits the events to InterChange Server. • `PropValue = LogError`: Logs an error but does not shut down. • `PropValue = Ignore`: Ignores these entries in the event table. This behavior is configurable via the `InDoubtEvents` connector property. Use the Notes field to describe exactly how the connector handles this feature.	__ Full __ Partial __ No __ N/A
Physical delete event	The connector creates an empty business object with the Delete verb, with key values populated and the rest of the attributes populated with CxIgnore, and sends the object to InterChange Server. See "Processing Delete events" on page 132.	__ Full __ Partial __ No __ N/A
RetrieveAll	The connector retrieves the entire hierarchical business object during subscription delivery. See "Retrieving application data" on page 187.	__ Full __ Partial __ No __ N/A

Table 163. Standard features for event notification (continued)

Category and name	Description	Supported?
Smart filtering	Duplicate events are not saved in the event store. Before storing a new event as a record in the event store, the event detection mechanism queries the event store for existing events that match the new event. The event detection mechanism does not generate a record for a new event in these cases: • The business object name, verb, key, status, and ConnectorId (if applicable) in a new event match those of another unprocessed event in the event store. • The business object name, key, and status for a new event match an unprocessed event in the event store. In addition, the verb for the new event is Update, and the verb for the unprocessed event is Create. • The business object name, key, and status for a new event match an unprocessed event in the event store. In addition, the verb in the unprocessed event in the event store is Create, and the verb in the new event is Delete. In this case, remove the Create record from the event store.	__ Full __ Partial __ No __ N/A
Verb stability	The connector should send a business object with the same verb that is in the event table. See "Getting the business object name, verb, and key" on page 184.	__ Full __ Partial __ No __ N/A

General standards

Table 164 lists general standards for connector behavior.

Table 164. General standards

Category and Name	Description	Supported?
Business Object		
Foreign key	There is no standard defined. If you use this property, check *Full* and describe how you use it. If you do not use this property, check *No*.	__ Full __ Partial __ No __ N/A
Foreign key attribute property	If this attribute property is set to true, the connector verifies that the value is a valid key. If the key is invalid, the connector returns `FAIL`. The connector assumes a foreign key is present in the application, and the connector should never try to create an object marked as a foreign key.	__ Full __ Partial __ No __ N/A
Key	There is no standard defined. If you use this property, check *Full* and describe how you use it. If you do not use this property, check *No*.	__ Full __ Partial __ No __ N/A
Max Length	There is no standard defined. If you use this property, check *Full* and describe how you use it. If you do not use this property, check *No*.	__ Full __ Partial __ No __ N/A
Required	There is no standard defined. If you use this property, check *Full* and describe how you use it. If you do not use this property, check *No*.	__ Full __ Partial __ No __ N/A
metadata driven design	The connector can support new business objects without recompiling because business object processing is based on metadata in the business object definition. See "Assessing support for metadata-driven design" on page 47.	__ Full __ Partial __ No __ N/A

Table 164. General standards (continued)

Category and Name	Description	Supported?
Loss of Connection to Application		
Connection lost on request processing	The connector detects the connection error when processing a business object request and shuts down. The connector logs a fatal error and sends a return code of APPRESPONSETIMEOUT so that email notification can be triggered. See "Handling loss of connection to an application" on page 78.	__ Full __ Partial __ No __ N/A
Connection lost on poll	The connector detects the connection error at the time of a poll call and shuts down. The connector logs a fatal error and sends a return code of APPRESPONSETIMEOUT so that email notification can be triggered. See "Handling loss of connection to an application" on page 78.	__ Full __ Partial __ No __ N/A
Connection lost while idle	Connector shuts down as soon as the connection to the application is lost. The connector logs a fatal error and sends a return code of APPRESPONSETIMEOUT so that email notification can be triggered.	__ Full __ Partial __ No __ N/A
Connector Properties		
ApplicationPassword	The connector should use this property value as the password to log in to the application.	__ Full __ Partial __ No __ N/A
ApplicationUser Name	The connector should use this property value as the user name to log in to the application.	__ Full __ Partial __ No __ N/A
UseDefaults connector property	If this connector property is set to true, when the connector processes a business object request with a Create verb, it calls the JCDK or CDK method initAndValidateAttributes().	__ Full __ Partial __ No __ N/A
Message Tracing		
General messaging	Messages that identify the business object handlers used for each object. Messages that log each time a business object is posted to Interchange Server, either from gotApplEvent() or consumeSync(). Messages that indicate each time a business object request is received. Guidelines for the trace messages at each trace level 0-5 follow. Note that the connector should deliver all the trace messages applicable at the level of tracing set and lower. See "Trace messages" on page 141.	__ Full __ Partial __ No __ N/A
Trace Level 0	0 - Message that identifies the connector version. No other tracing is done at this level.	__ Full __ Partial __ No __ N/A
Trace Level 1	1 - Status messages and identifying (key) information for each business object processed. A message is sent each time the pollForEvents() method is executed.	__ Full __ Partial __ No __ N/A
Trace Level 2	2 - Messages that identify the business object handlers used for each object the connector processes. Messages that log each time a business object is posted to InterChange Server, either from gotApplEvent()or consumeSync(). Messages that indicate each time a business object request is received.	__ Full __ Partial __ No __ N/A

Table 164. General standards (continued)

Category and Name	Description	Supported?
Trace Level 3	3 - Messages that identify the foreign keys being processed (if applicable). These messages appear when the connector has encountered a foreign key in a business object or when the connector sets a foreign key in a business object. Messages that relate to business object processing. Examples of this include finding a match between business objects, or finding a business object in an array of child business objects.	__ Full __ Partial __ No __ N/A
Trace Level 4	4 - Messages that identify application-specific information. Examples of this text include the values returned by the functions that process the application-specific information fields in business objects. Messages that identify entry or exit functions. These messages help trace the process flow of the connector. Messages that trace any thread-specific processing. For example, if the connector spawns multiple threads, a message should log the creation of each new thread.	__ Full __ Partial __ No __ N/A
Trace Level 5	5 - Messages that indicate connector initialization. The messages include the value of each configuration property that has been retrieved from InterChange Server. Messages that detail the status of each thread the connector spawns while it is running.The connector log file contains all statements executed in the application and the value of any variables that are substituted (where applicable). Messages for business object dumps. The connector outputs a text representation of a business object before it begins processing (showing the object the connector receives from the integration broker) and after it has processed the object (showing the object the connector returns to the integration broker).	__ Full __ Partial __ No __ N/A
Message tracing	Do not use the CDK method `generateMsg()` for tracing; instead, hard-code the message strings for trace messages.	__ Full __ Partial __ No __ N/A
Miscellaneous Features		
Java package names	All Java-based connectors should follow these package naming standards: com.CompanyName.connectors.ConnectorAgentPrefix Example: `com.crossworlds.connectors.XML`	__ Full __ Partial __ No __ N/A
Logging messages	The connector logs errors and other information that the user needs regardless of the trace level set for the system. See 139.	__ Full __ Partial __ No __ N/A
CDK method `logMsg()`	Always use the CDK method `generateMsg()` before calling `logMsg()`.	__ Full __ Partial __ No __ N/A
NT service compliance	To be NT service-compliant, do not use any method or function that points to `STDOUT`, for example, the `printf()` method in C++.	__ Full __ Partial __ No __ N/A
Transaction support	An entire business object request must be wrapped in a single transaction. All Create, Update, and Delete transactions for a top-level business object and all of its children should be wrapped in a single transaction. If any failure is detected during the life of the transaction, the whole transaction should be rolled back.	__ Full __ Partial __ No __ N/A
Special IBM CrossWorlds Values		
CxBlank processing	On a Create operation, the connector inserts an appropriate blank value for attributes with the value CxBlank. The blank value may be configurable or specific to the application. See “Handling the Blank and Ignore values” on page 171.	__ Full __ Partial __ No __ N/A

Table 164. General standards (continued)

Category and Name	Description	Supported?
CxIgnore processing	The connector does not set a value in the application for attributes that are passed in with the value CxIgnore when processing Create or Update verbs. See "Handling the Blank and Ignore values" on page 171.	__ Full __ Partial __ No __ N/A

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Burlingame Laboratory Director
IBM Burlingame Laboratory
577 Airport Blvd., Suite 800

Burlingame, CA 94010
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not necessarily tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

IBM
the IBM logo
AIX
CrossWorlds
DB2
DB2 Universal Database
Domino
Lotus
Lotus Notes
MQIntegrator
MQSeries
Tivoli
WebSphere

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

MMX, Pentium, and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others. WebSphere Business Integration Adapter Framework V2.4.0

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X