IBM WebSphere Business Integration Adapters
IBM WebSphere InterChange Server

Data Handler Guide

IBM WebSphere Business Integration Adapters
IBM WebSphere InterChange Server

Data Handler Guide

Note!
Before using this information and the product it supports, read the information in "Notices" on page 223.

19December2003

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 document, 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 Adapter 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 delivered data handlers and custom data handler capabilities.

Audience

This document is for consultants and customers. You should be familiar with business objects and metadata objects. To use the XML data handler, you should be familiar with XML documents, current XML standards, and with SAX (Simple API for XML). To use the EDI data handler, you should be familiar with EDI documents and current EDI standards. If you intend to expand the data handler library, you should be proficient in the Java programming language.

Related documents

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

You can install related documentation from 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 InterChange Server:

  http://www.ibm.com/websphere/integration/wicserver/infocenter

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

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

  http://www.ibm.com/software/webservers/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.
new term	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 inside 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 IBM WebSphere product pathnames are relative to the directory where the IBM WebSphere product is installed on your system.
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".
`%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.

New in this release

The recent revision history of this document is as described in the following sections.

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

The IBM WebSphere InterChange Server 4.2.2 release and the WebSphere Business Integration Adapter Framework 2.4 release provide the following changes to the data handlers:

- Both of these products now include a binary host data handler as part of the set of base data handlers. For more information about this data handler, see Chapter 9, "Binary host data handler," on page 155.
- Both of these products support the following changes to the Request-Response data handler:
  - The Request-Response data handler supports the conversion of both text and binary data. This is possible because the binary host data handler can be used in conjunction with the Request-Response data handler. For more information about the Request-Response data handler Chapter 5, "Request-Response data handler," on page 113.
  - The Request-Response data handler default child meta object definition MO_DataHandler_DefaultRequestResponseConfig is delivered with the data handlers in the default repository. This meta object definition can be used to configure the Request-Response data handler.
- Both of these products support the following changes to the XML data handler:
  - The XML data handler supports type substitution in business objects created from XML schema. For more information, see "Type substitution in business object definitions based on schema documents" on page 61.

New in WebSphere InterChange Server v4.2.1 and WebSphere Business Integration Adapter v2.3.1

The IBM WebSphere InterChange Server 4.2.1 release and the WebSphere Business Integration Adapter Framework 2.3.1 release now provide a new data handler, the Request-Response data handler. For more information, see Chapter 5, "Request-Response data handler," on page 113.

Both these products now support the following changes to the XML data handler:

- New way to specify names of XML elements and attributes
  - The business-object attribute that represents an XML element must now contain the `elem_name` tag in its application-specific information. For more information, see "For XML elements" on page 49 (for XML elements defined in DTDs) or "For XML elements" on page 72 (for XML elements defined in schema documents).
  - The business-object attribute that represents an XML attribute must now contain the `attr_name` tag in its application-specific information (in addition to the `type=attribute` tag). For more information, see "For an XML attribute" on page 50 (for XML attributes defined in DTDs) or "For an XML attribute" on page 75 (for XML attributes defined in schema documents)

Note: The `elem_name` and `attr_name` tags replace previous syntax, which required only the name of the XML element or attribute in the business-object attribute's application-specific information. The XML data handler still supports the old syntax for backward compatibility with existing business object definitions. However, the XML ODA uses the new syntax when it generates business object definitions.

- New way to specify the namespaces of a schema document

  The business object definition must now contain the `target_ns` tag in its application-specific information to identify the target namespace. For more information, see "Schema namespaces" on page 64.

  Note: The `target_ns` tag replaces previous syntax, which required that the business object definition contain attributes that defined the prefixes of each namespace. These attributes indicated whether they represented the default namespace or a prefixed namespace with the `type=defaultNS` or `type=xmlns` tags (respectively) in their application-specific information. The XML data handler still supports the old syntax for backward compatibility with existing business object definitions. However, the XML ODA uses the new syntax when it generates business object definitions.

- New way to specify qualified XML element and attribute names

  The XML data handler now recognizes the following XML structures:

  – The `elementFormDefault` and `attributeFormDefault` attributes of the schema element

    Based on the values of these schema attributes, the `elem_fd` and `attr_fd` tags in the application-specific information of the business object definition indicate whether XML element and attribute names are qualified. For more information, see "Qualified component names" on page 69
    application-specific information

  – The `form` attribute of an XML element or attribute

    Based on the value of this attribute, the `elem_fd` and `attr_fd` tags in the application-specific information of the business-object attribute indicate whether the particular XML element or attribute name is qualified.

  Note: The `elem_fd` and `attr_fd` tags replace previous syntax, which required that some business object definitions contain namespace prefixes in the names of the attributes that represented XML elements or attributes. The business object definition no longer stores namespace prefix information. The XML data handler still supports the old syntax for backward compatibility with existing business object definitions. However, the XML ODA uses the new syntax when it generates business object definitions.

- The XML data handler now supports the following XML structures of a schema document:

  – Multiple schema namespaces (the `import` element) by providing the `elem_ns` and `attr_ns` tags in the application-specific information of business-object attributes that represent XML elements and attributes, respectively. For more information, see "Schema namespaces" on page 64.

  – Some restrictions on simple-content and complex-content complex types (the `restriction` element). For more information, see "Supported schema-document structures" on page 78.

New in WebSphere InterChange Server v4.2 and WebSphere Business Integration Adapter v2.2.0

The IBM WebSphere InterChange Server 4.2 release and the WebSphere Business Integration Adapter Framework 2.2.0 release provide the following changes to data handlers:

- These releases no longer use the ″CrossWorlds″ name to describe an entire system or to modify the names of components or tools. For example ″IBM CrossWorlds XML Data Handler″ is now ″IBM WebSphere Business Integration Data Handler for XML,″ and ″CrossWorlds InterChange Server″ is now ″WebSphere InterChange Server.″
- You can now install data handlers as part of either of the these IBM WebSphere products:
  - IBM WebSphere InterChange Server: *all* IBM-delivered data handlers are part of this product.
  - IBM WebSphere Business Integration Adapters: *only* the XML data handler is part of this product.

  For more information, see “Installing data handlers” on page 21.
- The WebSphere InterChange Server 4.2 release changes the default configuration of the `MO_Server_DataHandler` meta-object. This meta-object now contains only a dummy attribute. Therefore, by default the Server Access Interface process (within InterChange Server) does not support any data handlers. For information on how to configure this meta-object to provide data-handler support for the Server Access Interface process, see “MO_Server_DataHandler meta-object” on page 25.

New in WebSphere Business Integration Adapter Framework v2.1

The WebSphere Business Integration Adapter Framework 2.1.0 release provides a new XML ODA to create business object definitions from the following XML data models:

- XML Document type definitions (DTDs)

  The previous version of the XML data handler performed conversion between XML DTDs and business object definitions. It used two external tools to create business object definitions from DTDs. This functionality is now provided by the XML ODA. For more information, see “XML documents that use DTDs” on page 42.
- XML Schema documents

  Support for conversion between schema documents and business object definitions is new with the XML ODA. For more information, “XML documents that use schema documents” on page 54.

For information on the use of the XML ODA, see “Using an XML ODA to create business object definitions” on page 80.

New in WebSphere Business Integration Adapter Framework v2.0.1

The WebSphere Business Integration Adapter Framework 2.0.1 release provides an internationalized version of many adapters and data handlers. These internationalized products have been localized for the English and Japanese locales (A locale includes culture-specific conventions and a character code set.). For details in this guide, see the following sections:

- The Data Handler API now provides methods to allow a custom data handler to access the locale and character encoding of the data handler environment:
  - To access the locale: getLocale() and setLocale()
  - To access the character encoding: getEncoding() and setEncoding()
- For an overview of how to internationalize a data handler, see "An internationalized data handler" on page 190.

New in WebSphere Business Integration Adapter Framework v2.0

Data handlers are now supported with the WebSphere MQ Integrator integration broker. For more information about this integration broker, see the *Implementation Guide for WebSphere MQ Integrator Broker* in the WebSphere Business Integration Adapters documentation set. Data handlers continue to support InterChange Server as an integration broker.

New in release 4.1.0

- The XML data handler now populates prolog information such as the DOCTYPE and XML declaration, if there are attributes that correspond to those elements in the business object definition.
- The Delimited data handler can now take a String with multiple characters as the delimiter.

New in release 4.0.1

- The `tpi_rnif` data handler has been added to the `CwDataHandler.jar` for TPI support of the Chem eStandards.

Significant performance improvements have been made to the XML data handler. Most of these are transparent. Some are not:

- Business object attributes representing XML elements or XML attributes with content that requires escape processing must now use the `escape=true` application-specific information. In previous versions of the data handler, all attributes were escape processed by default. In this latest version of the XML data handler, an attribute will not be escape processed unless it contains the `escape=true` application-specific information. If an attribute represents an XML element whose value contains single quotes, double quotes or the &, <, or > characters, then the attribute requires escape processing. This new application-specific information (`escape=true`)must be placed at the end of any existing text. For example:

```
[Attribute]
Name=Data
Type=String
AppSpecificInfo=Price;type=pcdata;escape=true
[End]
```

- A new XML data handler child meta-object attribute, `InitialBufferSize`, has been created to define the initial size of the buffer that is used when converting business objects to XML. Set this value to the size, in bytes, of your XML business objects. Setting this value to a high number will speed the conversion of business objects to serialized XML. The default value is 2 MB.
- A new XML data handler child meta-object attribute, `UseNewLine`, has been created. Set this attribute to `true` if you want each tag in the output XML to be on a new line. (The XML data handler adds extra content in the form of line feeds and carriage returns to the XML document.) Set to `false` if you do not want to alter the XML output.

New in release 4.0.0

- The Xerces Parser from Apache is now the default parser for the XML Data Handler. Instructions are provided for using the SAX Parser from IBM instead of the default.
- The Edifecs SpecBuilder utility is now the preferred XML data handler tool for creating business object definitions.
- The attribute `OmitObjectEventId` has been added to the FixedWidth and Delimited data handlers.

Part 1. Getting started

Chapter 1. Data handler overview

This chapter introduces WebSphere business integration system data handlers. A data handler is responsible for converting business objects into serialized data and for converting serialized data into business objects. This serialized data is in an application-readable format (string or input stream). This chapter contains the following sections:

What is a data handler?

A *data handler* is a Java class instance that converts between a particular serialized format and a business object. Data handlers are used by components of a business integration system that transfer information between a WebSphere business integration broker and some external process. Table 1 shows the components that handle transfer of information between a WebSphere business integration broker and an external process.

Table 1. Components that transfer information in the WebSphere business integration system

Component	Purpose	For more information
Adapter	Handles transfer of information between a WebSphere business integration broker and an external process such as an application or technology. **Note:** The adapter uses a runtime component called a *connector* to actually handle the transfer of information between an integration broker and an application (or technology). These external processes identify events that occur within them by sending an event record to an event store. The adapter detects events in this event store. When it finds a triggering event, the adapter creates a business object that represents the event and sends this event *asynchronously* to the business integration broker. This business object contains data and a verb to indicate the type of event (such as Create or Update).	For IBM-delivered adapters: see the individual adapter guides. For a custom adapter, see the *Connector Development Guide for Java* or the *Connector Development Guide for C++* (depending on the language in which the connector is implemented) for information about custom connectors.

Table 1. Components that transfer information in the WebSphere business integration system (continued)

Component	Purpose	For more information
Access client	Handles transfer of information between the InterChange Server integration broker and some external process such as a servlet within a web server. An access client is an external process that uses the Server Access Interface to communicate directly with InterChange Server. When this component receives some information that needs to be transferred, it creates a business object that represents the event and sends this event *synchronously* to a collaboration within InterChange Server. As with the adapter, the business object contains data and a verb to indicate the type of event (such as Create or Update).	Server Access Interface Development Guide

As Table 1 shows, the task of both these components (connector and access client) is to transfer information between a broker and an external process, as follows:

- To send information to an integration broker, these components format it in a business object.
- To send information to an external process, these components format it in its native serialized format.

Often, the external process uses some common format such as XML for its native serialized data. Rather than have every adapter (or access client) handle the transformation between these common formats and business objects, the WebSphere business integration system provides several IBM-delivered data handlers. In addition, you can create custom data handlers to handler conversion between your own native format. The adapter (or access client) can then call the appropriate data handler to perform the data conversion based on the Multipurpose Internet Mail Extensions (MIME) type of the serialized data.

Note: A data handler is implemented in a Java class named `DataHandler`. This class is an abstract class, which the data-handler developer extends to implement a data handler instance. For more information, see "Extending the data handler base class" on page 171.

The section provides the following information about data handlers:

- "IBM-delivered data handlers"
- "Data-handler meta-objects" on page 6
- "Contexts for calling data handlers" on page 6

IBM-delivered data handlers

IBM delivers data handlers in the Java archive (`jar`) files shown in Table 2. These `jar` files reside in the `DataHandlers` subdirectory under the product directory.

Table 2. IBM-delivered data-handler jar files

Contents	Description	Data-handler jar file
Base data handlers	Text-based data handlers and data handlers specific to some IBM-delivered adapters	`CwDataHandler.jar`
Special data handlers	XML data handler	`CwXMLDataHandler.jar`
	EDI data handler	`CwEDIDataHandler.jar`
Custom data handlers	Data handlers that you implement	`CustDataHandler.jar`

Base data handlers

The base data-handler file, `CwDataHandler.jar`, contains most of the IBM-delivered data handlers. This file resides in the `DataHandlers` subdirectory of the product directory. Table 3 shows the base data handlers that this base data-handler file contains.

Table 3. Base data handlers in the base data-handler file

Data handler	MIME type	For more information
Request-Response Data Handler	`text/requestresponse`	Chapter 5, "Request-Response data handler," on page 113
FixedWidth Data Handler	`text/fixedwidth`	Chapter 6, "FixedWidth data handler," on page 131
Delimited Data Handler	`text/delimited`	Chapter 7, "Delimited data handler," on page 139
NameValue Data Handler	`text/namevalue`	Chapter 8, "NameValue data handler," on page 147
Binary Host Data Handler	N/A	Chapter 9, "Binary host data handler," on page 155

Note: This manual describes the text data handlers that Table 3 lists. The base data-handler file also contains several data handlers specific to certain IBM-delivered adapters. If an IBM adapter uses a special data handler, its adapter guide describes the installation, configuration, and use of its data handler.

Special data handlers

IBM makes separate installers available for a few data handlers. In order to install these special data handlers, you must follow the steps provided in the *Installation Guide for WebSphere Business Integration Adapters*.

The separation of a data handler from the base data-handler file allows many adapters to use the data handler without incurring the overhead of storing the other data handlers that reside in the base data-handler file. Table 4 shows the data handlers for which IBM provides separate installers and separate jar files.

Table 4. IBM-delivered data handlers with separate jar files

Data handler	Data-handler jar file	MIME type	For more information
XML Data Handler	`CwXMLDataHandler.jar`	`text/xml`	Chapter 3, "XML data handler," on page 31
EDI Data Handler	`CwEDIDataHandler.jar`	`edi`	Chapter 4, "EDI data handler," on page 89

Custom data handlers

If the IBM-delivered data handlers to *not* handle the conversion of serialized data to a business object, you can create your own custom data handler. The `CustDataHandler.jar` file is intended to hold any custom data handlers that you might develop. This file resides in the `DataHandlers` subdirectory of the product directory. For information about how to create a custom data handler, see Chapter 10, "Creating a custom data handler," on page 167.

Note: To assist in develop of custom data handlers, IBM also delivers source code for the FixedWidth, Delimited, and NameValue data handlers as sample code. For more information, see "Sample data handlers" on page 169.

Data-handler meta-objects

A connector or Server Access Interface process (if InterChange Server if your integration broker) instantiates a data handler based on the MIME type of an input file or the MIME type specified in a business object request.

A data-handler meta-object is a hierarchical business object that can contain any number of child objects. The data-handler configuration information is arranged in the following hierarchy:

- The *top-level* meta-object contains information about the MIME types that the different data handlers can support. Each top-level attribute is a cardinality 1 attribute referencing a child meta-object for a data handler instance. Each attribute represents one MIME type and indicates which data handler can manipulate it.
- The *child* meta-object contains the actual configuration information for a particular data handler. Each attribute represents a configuration property and provides information such as its default value and type.

Note: A data handler is not required to use meta-objects to hold configuration information. However, *all* IBM-delivered data handlers are designed to use meta-objects for their configuration information.

Data-handler meta-objects allow a connector or Server Access Interface process (if InterChange Server if your integration broker) to instantiate a data handler based on the MIME type of an input file or the MIME type specified in a business object request. To configure a data handler, you must ensure that its meta-objects are correctly initialized and available to the callers (a connector or an access client).

Note: Each IBM-delivered data handler uses configuration properties that are defined in data-handler meta-objects. However, a custom data handler might or might not use meta-objects for its configuration properties. For more information, see "Using data-handler meta-objects" on page 171.

Contexts for calling data handlers

As Table 1 on page 3 describes, a component that needs to transfer data in the WebSphere business integration system can invoke a data handler. Table 5 provides additional information about the components that can invoke a data handler.

Table 5. Context for calling data handlers

Component	Type of event communication	Type of flow	Software that invokes the data handler
Adapter	Asynchronous	Event-triggered flow	Connector

Table 5. Context for calling data handlers (continued)

Component	Type of event communication	Type of flow	Software that invokes the data handler
Access client (InterChange Server integration broker only)	Synchronous	Call-triggered flow	Server Access Interface (within InterChange Server)

As Table 5 shows, in an event-triggered flow, an adapter calls a data handler directly. In a call-triggered flow, an external process that uses the Server Access Interface (called an access client) initiates a call to the data handler. A data handler operates the same whether it is called directly by an adapter or indirectly by an access client. These contexts are described in the next sections.

Data handlers in a connector context

In an event-triggered flow, the runtime component of an adapter, called the *connector*, interacts directly with a data handler to convert data.

Note: For IBM-delivered adapters, see the individual adapter guides. For a custom adapter, see the *Connector Development Guide for Java* or the *Connector Development Guide for C++*, depending on the language in which the adapter is implemented. These guides are part of the WebSphere Business Integration Adapters documentation set.

When a connector calls a data handler, the data handler runs as part of the connector process. Figure 1 illustrates the data handler in the context of a connector.

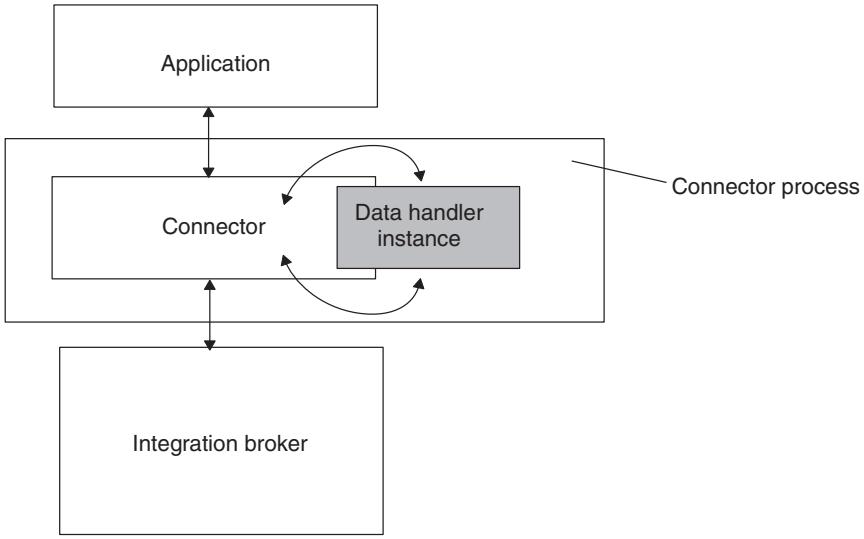

Figure 1. Data handler in the context of a connector

The data conversion reflects the business object requirements and the direction of the flow:

- The connector invokes the data handler for a business-object-to-string conversion when it handles business object request processing.
- The connector invoke the data handler for a string-to-business-object conversion when it handles event notification.

Connector business-object-to-string conversion: For a business-object-to-string conversion, the connector calls the data handler, passing it a business object. The data handler uses the information in the business object and the business object definition to create a stream or string of data. This stream or string of data is in the format associated with the data handler, usually of a particular MIME type. Business-object-to-string conversion is useful when the connector receives information from an integration broker in the form of a business object. The connector must then send the information in the business object to its application (or technology) as serialized data.

Figure 2 illustrates the data handler in the context of a connector when the data handler performs a business-object-to-string conversion.

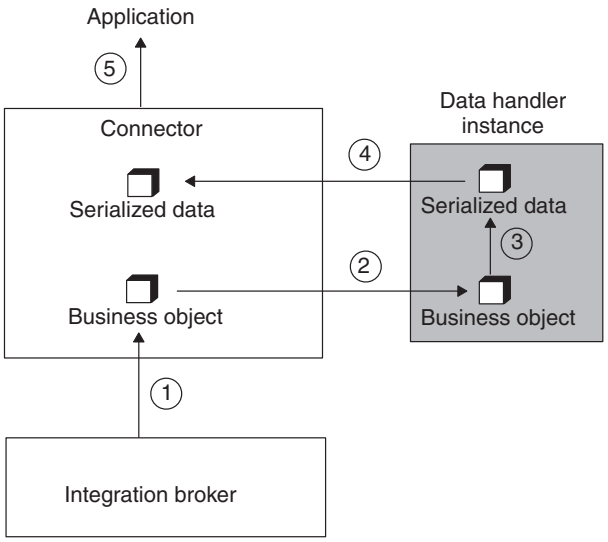

Figure 2. Business-object-to-string conversion in the connector context

1. The connector receives a business object from an integration broker.
2. The connector creates an instance of a data handler to handle the business object (using the `createHandler()` static method in the `DataHandler` base class).

   For more information about how a connector instantiates the data handler, see "Instantiation in the context of a connector" on page 16.
3. The connector requests a business-object-to-string conversion by calling one of the following data-handler methods:
   - `getStreamFromBO()`
   - `getStringFromBO()`
   - `getByteArrayFromBO()`

   Into this method, the connector sends the business object as an argument. The data handler serializes the business object into the requested data format.
4. The data handler returns the serialized data to the connector.
5. The connector writes the serialized data to the destination, which could be an email, a file, or an HTTP connection.

Connector string-to-business-object conversion: For a string-to-business-object conversion, the connector calls the data handler, passing it the serialized data and its associated MIME type object. The data handler receives a stream or string of data. The data handler uses the information in the data stream to create, name, and populate a business object instance of the specified type. String-to-business-object

conversion is useful when the connector needs to send an event to an integration broker. The application sends this event as serialized data having a particular MIME type, to the connector.

Figure 3 illustrates the data handler in the context of a connector when the data handler performs a string-to-business-object conversion.

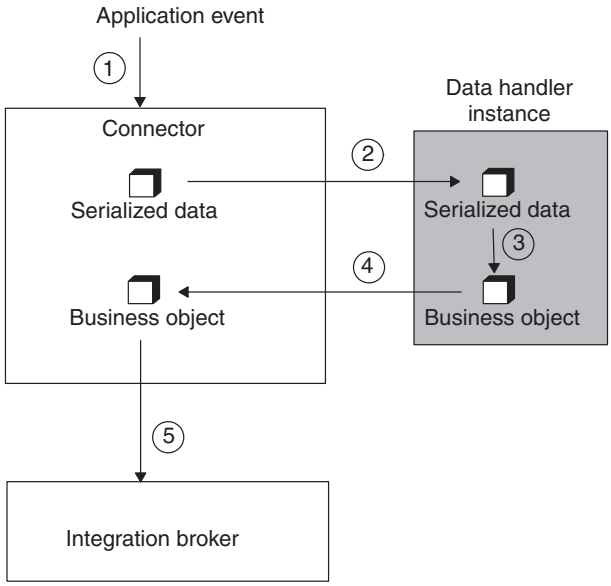

Figure 3. String-to-business-object conversion in the connector context

1. The connector detects an application event. The event may be in the form of an email, a text file, an XML document, or any other common format for which a data handler exists.
2. The connector creates an instance of a data handler to handle the event (using the `createHandler()` static method in the `DataHandler` base class).

   For more information about how a connector instantiates the data handler, see "Instantiation in the context of a connector" on page 16.
3. The connector sends in the serialized data as an argument to the `getBO()` method of the data-handler instance. The data handler builds an instance of a business object.

   The connector might also specify the business object to which the `getBO()` method converts the data. Some connectors specify the business object type; others assume that the data handler can extract the business object type from the serialized text. The data handler parses the data and populates the attribute values for the business object based on the serialized data.
4. The data handler returns the business object to the connector.
5. The connector sends the business object to the integration broker.

Data handlers in the context of the Server Access Interface

In a call-triggered flow, an access client interacts with a data handler to convert data. An *access client* is an external process that uses the Server Access Interface to interact with InterChange Server. When an access client calls either the `ItoExternalForm()` or `IcreateBusinessObjectFrom()` method of the Server Access Interface API, it initiates a call to a data handler. The Server Access Interface, which runs as part of the InterChange Server process, actually invokes the data handler.

InterChange Server

The Server Access Interface is an API that allows an access client to execute a collaboration inside InterChange Server. This interface is available for use *only* when InterChange Server is the integration broker. For more information on this interface and on access clients, see the *Access Development Guide* in the IBM WebSphere InterChange Server documentation set.

An access client might be a servlet that handles a request from a client browser. The request might be a request for data, an order request, or another type of business-to-business transaction. As another example, an access client might be a C++ or Java program that uses the Server Access Interface to access InterChange Server and exchange data with another application.

When an access client initiates a call that requires a data handler, the data handler runs as part of InterChange Server the process. Figure 4 illustrates the data handler in the context of the Server Access Interface. In this example, the access client is a Web server and servlet.

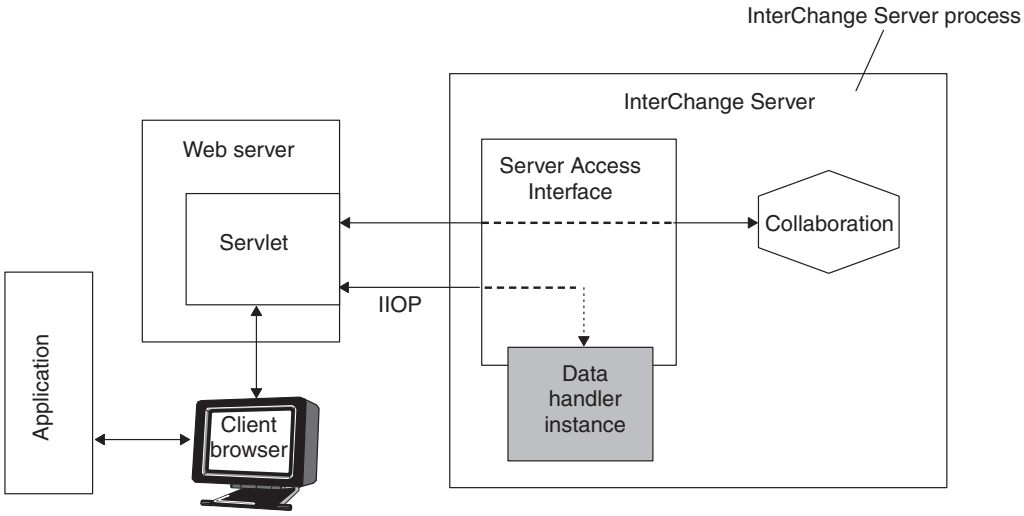

Figure 4. Data handler in the context of the Server Access Interface

The data conversion reflects the business object requirements and the direction of the flow:

- Receiving a business object from InterChange Server requires that the access client ask the data handler for a business-object-to-string conversion.
- Sending data to InterChange Server requires that the access client ask the data handler for a string-to-business-object conversion.

Server Access Interface business-object-to-string conversion: For a business-object-to-string conversion, the data handler receives a business object as the result of the execution of a collaboration. The data handler uses the information in the business object to create a stream or string of data. This data is in the format associated with the data handler, usually of a particular MIME type. The access client often sends the resulting business object to the application as serialized data.

Figure 5 illustrates the data handler in the context of the Server Access Interface when the data handler performs a business-object-to-string conversion for an access client.

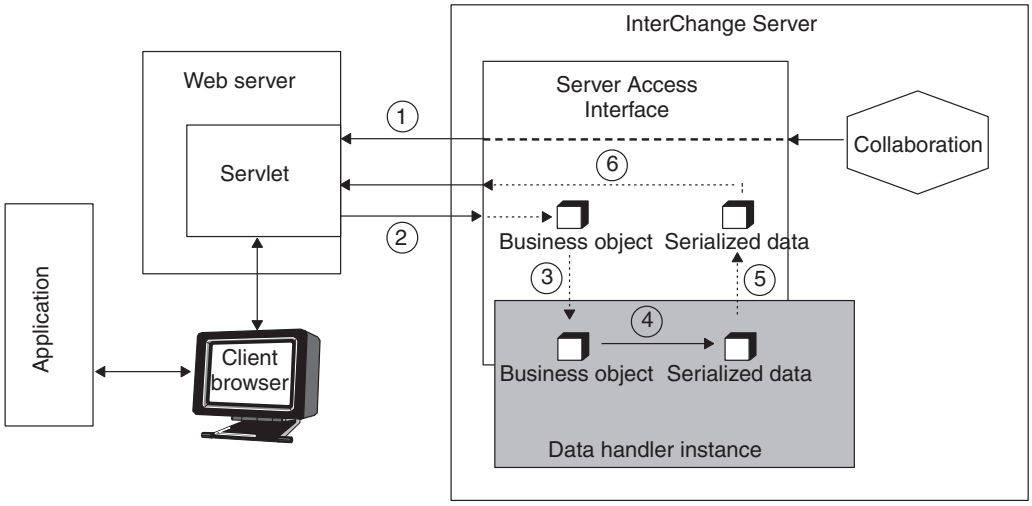

Figure 5. Business-object-to-string conversion in the Server Access Interface context

1. The collaboration returns the requested data or the results of requested actions to the access client.
2. To convert the business object to the required format, the access client sends the business object as an argument to the `ItoExternalForm()` method of the Server Access Interface.
3. The Server Access Interface takes the following actions:
   - Create an instance of a data handler to handle the conversion (using the `createHandler()` static method in the `DataHandler` base class).
   - Send in the serialized data as an argument to one of the following data-handler methods:
     - `getStreamFromBO()`
     - `getStringFromBO()`
     - `getByteArrayFromBO()`
4. The data handler parses the business object to create the serialized data.
5. The data handler returns the serialized data to the Server Access Interface.
6. The Server Access Interface returns the serialized data to the access client.

Server Access Interface string-to-business-object conversion: For a string-to-business-object conversion, the data handler receives a stream or string of data. The data handler uses the information in the data stream to create, name, and populate a business object instance of the specified type. String-to-business-object conversion is useful when the access client needs to send a business object to a collaboration in InterChange Server. The access client sends the serialized data, usually having a particular MIME type, to the data handler.

Figure 6 illustrates the data handler in the context of the Server Access Interface when the data handler performs a string-to-business-object conversion for an access client.

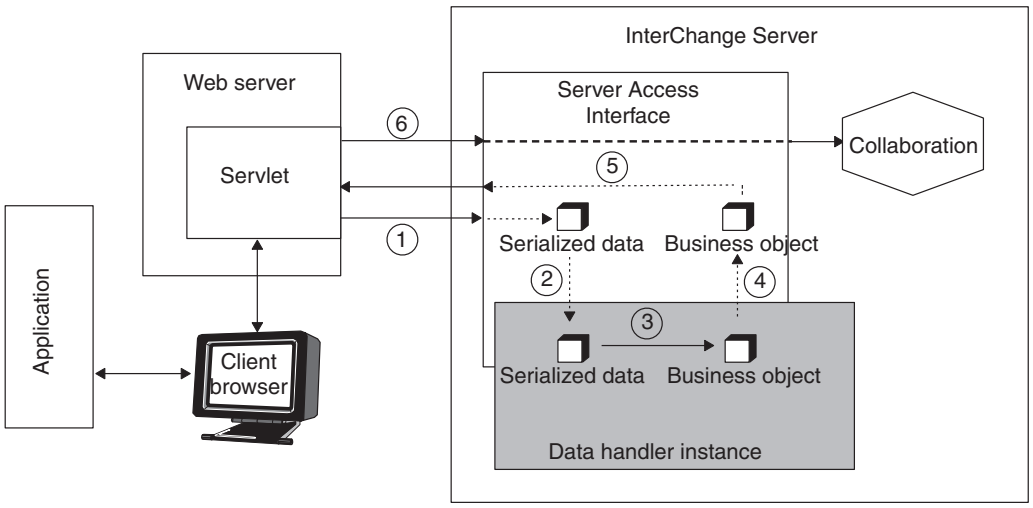

Figure 6. String-to-business-object conversion in the Server Access Interface context

1. To convert serialized data to a business object, the access client sends the serialized data as an argument to the `IcreateBusinessObjectFrom()` method of the Server Access Interface.
2. The Server Access Interface takes the following actions:
   - Create an instance of a data handler to handle the conversion (using the `createHandler()` static method in the `DataHandler` base class).
   - Send in the serialized data as an argument to the `getBO()` method of the data handler instance.
3. The data handler builds an instance of a business object.

   The data handler parses the data and populates the attribute values for the business object based on the serialized data.
4. The data handler returns the business object to the Server Access Interface.
5. The Server Access Interface returns the business object to the access client.
6. The access client calls a collaboration that uses the business object data in a business process.

Data handler instantiation

Data handlers are implemented as a library of classes that a connector or the Server Access Interface (for an access client communicating with the InterChange Server integration broker) can use. The `DataHandler` base class is an abstract class. Therefore, to instantiate a data handler, you must instantiate one of the `DataHandler` subclasses. Each data handler, either an IBM-delivered data handler or a custom data handler, is a subclass of the `DataHandler` base class. The method to instantiate an instance of a data handler is `createHandler()`.

The `createHandler()` method uses information in the *data-handler meta-objects* to determine which data handler to instantiate and how to initialize this data handler. A data-handler meta-object is a hierarchical business object that can contain any number of child objects. The data-handler configuration information is arranged in the following hierarchy:

- The *top-level* meta-object contains information about the MIME types that the different data handlers can support. Each top-level attribute is a cardinality 1 attribute that references the child meta-object for a data handler instance. Each

attribute represents one MIME type and its attribute type indicates the child meta-object for the data handler that can manipulate this MIME type.
- The *child* meta-object contains the actual configuration information for a particular data handler. Each attribute represents a configuration property and provides information such as its default value and type.

Note: A data handler is not required to use meta-objects to hold configuration information. However, *all* IBM-delivered data handlers are designed to use meta-objects for their configuration information.

The `createHandler()` method takes the following steps to instantiate a data handler:

- "Identifying the data-handler class"
- "Setting data-handler configuration properties" on page 15
- "Setting the business-object prefix" on page 15

Identifying the data-handler class

For a data handler to be created, an implementation of the `DataHandler` base class must be instantiated. The data-handler instantiation method derives the name of this data-handler class from one of two values, which is passed in as an argument to the `createHandler()` method:

- The class name of the data handler to instantiate
- The MIME type of the data to convert

Using a class name

If the data-handler caller passes in a class name as an argument, `createHandler()` instantiates a data handler of that class name. It looks for the specified class in the following places:

1. `CwDataHandler.jar` file
2. `CwXMLDataHandler.jar` file
3. `CwEDIDataHandler.jar` file
4. `CustDataHandler.jar` file
5. Elsewhere in the `CLASSPATH`

If the caller provides *only* the class name for the data handler, `createHandler()` neither looks for data-handler meta-objects nor sets configuration properties from these objects. Therefore, a data handler instantiated in this way does *not* require meta-objects. For more information on whether a custom data handler should use meta-objects, see "Using data-handler meta-objects" on page 171.

Using a MIME type

If the data-handler caller does *not* pass in the class name as an argument, the `createHandler()` method requires a value for a MIME type. When a calling component (connector or access client) passes a MIME type, `createHandler()` 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: For more information on meta-objects, see "Configuring data handlers" on page 24. For more information on whether a custom data handler should use meta-objects, see "Using data-handler meta-objects" on page 171.

To derive a class name from a specified MIME type, the `createHandler()` method takes the following steps:

1. Convert the MIME type to a MIME-type string

   When the createHandler() method searches the top-level data-handler meta-object, it converts any non-alphanumeric characters, such as a hyphen (-), period (.) or slash (/), to an underscore (_). For example, if the MIME type is text/html, createHandler() parses the type to the string text_html.

   The createHandler() method performs this conversion of non-alphanumeric characters in stages, so that matches of MIME type names that contain periods can occur. However, Business Object Designer does not allow the period character in attribute names. Therefore, IBM recommends that you do *not* include them in the MIME type names.

   You can create unique MIME type/subtype combinations to indicate variations on a particular MIME type. In the MIME-type name, you separate the MIME type and subtype with a non-alphanumeric character (such as a hyphen or underscore). However, because createHandler() replaces any non-alphanumeric character with an underscore, IBM recommends that you use only an underscore to separate the MIME type and subtype. If the MIME type is text/xml-sgml, the method converts the type to the string text_xml_sgml.
2. Obtain the name of the top-level data-handler meta-object from a static property in the DataHandler base class. In this top-level meta-object, createHandler() looks for an attribute that matches the MIME-type string of the data to convert.
3. If createHandler() locates a matching MIME-type string in the top-level meta-object, createHandler() takes the following steps:
   a. If the caller supplied a value for the business-object prefix (an optional third argument) to createHandler(), interpret this value as a MIME subtype and append it to the MIME type to create a MIME-type string of the form:

      MIMETypeString_BOPrefix

      If no attribute of this name exists, createHandler() looks for an attribute that matches *only* the MIME type. For example, if the caller passes in a MIME type of edi and a business-object prefix of x12, createHandler() looks for an attribute in the top-level meta-object named "edi_x12". If no attribute by this name exists, createHandler() looks for an attribute named "edi".
   b. Obtain the associated child data-handler meta-object, which contains the configuration properties for the data handler instance to use (for example, a possible configuration property might identify which character to use for the data handler delimiter).
   c. Retrieve the value of the ClassName attribute from the child meta-object.
      - If the ClassName attribute contains a value, createHandler() instantiates a data handler of this class name. Part of the process of implementing a data handler is the creation of the associated child data-handler meta-object. At this time, the developer who implements the data handler can add the class name to the ClassName attribute of the child meta-object. This ClassName value is required if the class name is different from the default class name, as described in 4a.
      - If this ClassName attribute does *not* contain a value, createHandler() builds a class name for the data handler it instantiates, as described in 4a.
4. If the createHandler() method does *not* find a matching MIME-type string in the top-level meta-object, it builds a class name of the data handler that it will instantiate, as follows:
   a. Append the MIME-type string to the name of the base data handler package:

com.crossworlds.DataHandlers

For example, if the MIME-type string is text_html, the resulting string is:

com.crossworlds.DataHandlers.text.html

b. If the method can locate the generated class name on the CLASSPATH, it instantiates this class. It looks for this class in the following places:
   - CwDataHandler.jar file
   - CustDataHandler.jar file
- Elsewhere in the CLASSPATH

Using a class name and MIME type

If the caller provides *both* the class name and the MIME type, createHandler() takes the following actions:

- Create a data handler of the specified class. To locate this class, the data handler looks in the directories listed in "Using a class name" on page 13.
- Use the child meta-object associated with the MIME type to initialize the configuration options of this data handler. For information on how createHandler() determines the child meta-object from the MIME type, see "Using a MIME type" on page 13

In other words, when the caller provides a class name, this class name *overrides* the class name specified in the ClassName attribute of the child meta-object.

Setting data-handler configuration properties

All IBM-delivered data handlers (see Table 3 and Table 4) are designed to use data-handler meta-objects for their configuration information. A data-handler meta-object is a hierarchical business object:

- In the top-level data-handler meta-object, each attribute is identified with a MIME type and represents a child data-handler meta-object.
- The child data-handler meta-object contains configuration information for the data handler associated with the MIME type.

The IBM-delivered data handlers use the configuration information in its associated a child data-handler meta-object to initialize its properties. Therefore, IBM delivers a child meta-object for each of its delivered data handlers (see Table 10).

Note: For a description of data-handler meta-objects, see "Configuring data handlers" on page 24.

After the createHandler() method instantiates the data handler instance, it calls a special protected method, setupOptions(), to initialize the data handler's configuration with the values in the appropriate child data-handler meta-object.

Note: A custom data handler is not required to use meta-objects to initialize its configuration. If the data handler has an associated child meta-object, the createHandler() method does not call setupOptions() for the data handler. For more information, see "Setting up other business objects" on page 189.

For more information on meta-objects, see "Configuring data handlers" on page 24.

Setting the business-object prefix

The createHandler() method can accept an optional business-object prefix as its third argument. It uses this argument to determine the MIME type name (see

"Using a MIME type" on page 13). After `createHandler()` has instantiated the data handler instance and set its configuration properties, its final task is to set the value of the `BOPrefix` configuration option (if one exists) in the data handler to the value of this third argument.

Note: Any data handler that uses a `BOPrefix` configuration option (such as the XML data handler) prepends this prefix to business-object names.

The data handler can prepend this prefix to the names of any business objects that it creates (during string-to-business-object conversion). It puts an underscore (_) between the prefix and the business-object name. For example, a connector might invoke the XML data handler with the following `createHandler()` call:

```
createHandler(null, "text/xml", "UserApp");
```

The `createHandler()` method instantiates the XML data handler and set its `BOPrefix` attribute to "`UserApp`". When the XML data handler creates a `Customer` business object, it:

- Obtains the business object name from the serialized data
- Prepends the "`UserApp`" prefix to the business object name

The resulting business-object name is "`UserApp_Customer`".

Calling the data handler

Regardless of the context in which a data handler is called, the data handler is instantiated by the `createHandler()` method. How that method is called in each context is as follows:

- A connector *explicitly* calls `createHandler()` to instantiate a data handler.
- An access client calls `createHandler()` *implicitly*; the Server Access Interface actually calls `createHandler()` when the access client initiates a data-handler call through one of the following Server Access Interface methods: `ItoExternalForm()` or `IcreateBusinessObjectFrom()`.

Instantiation in the context of a connector

Figure 7 shows an example of data handler instantiation when the data handler is called in the context of a connector.

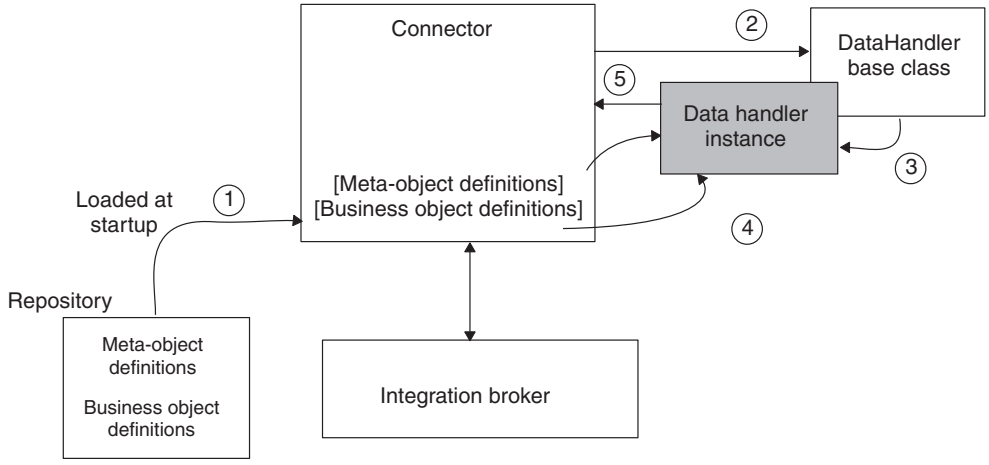

Figure 7. Data handler instantiation in the connector context

To instantiate a data handler called in the context of a connector, the connector takes the following steps:

1. Meta-objects are read into memory when the connector starts up, as long as the meta-objects are included on the list of supported objects for the connector.

   Meta-objects are stored in the repository along with business objects. The meta-object definitions, like business object definitions, must exist in memory for the data handler to be able to access them. When these meta-objects are in memory as part of the connector process, the meta-objects are accessible by a data handler called in the context of a connector.
2. The connector calls the `setConfigMOName()` static method in the `DataHandler` base class to set a static property in the data handler base class to the name of the top-level meta-object for the data handler.

   This top-level meta-object is the connector meta-object (`MO_DataHandler_Default` by default). The top-level meta-object must be part of the connector's supported objects list.

   Note: How the connector obtains the name of the data handler top-level meta-object is determined as part of the connector design. For more information, see "Configuring a connector" on page 189.
3. The connector calls the `createHandler()` static method in the `DataHandler` base class to create an instance of the `DataHandler` base class that performs the required data conversion. The name of the class to be created is determined in one of two ways:
   - If the connector passes in a class name as an argument, the `createHandler()` method instantiates a data handler of that class name. A connector can explicitly specify a class name when it calls `createHandler()`.
   - If the MIME type is passed in instead of the class name, `createHandler()` derives the class name from the MIME type.

     The `createHandler()` method converts the MIME type to a MIME-type string and obtains the name of the data handler's top-level meta-object from a static property in the `DataHandler` base class. From this top-level meta-object, `createHandler()` obtains the name of the child meta-object for the data handler. This child meta-object contains configuration information, including the name of the class to instantiate. For information on how this derivation occurs, see "Identifying the data-handler class" on page 13.
4. The data handler performs the required data conversion. The connector agent calls the appropriate `DataHandler` method to perform the required conversion:
   - The `getBO()` method for a string-to-business-object conversion.
   - The `getStringFromBO()` method for a business-object-to-string conversion or the `getStreamFromBO()` method for a business-object-to-stream conversion.
5. The data handler returns the appropriate format to the connector agent.

Instantiation in the context of the Server Access Interface

Figure 8 shows an example of data handler instantiation when the data handler is called in the context of the Server Access Interface.

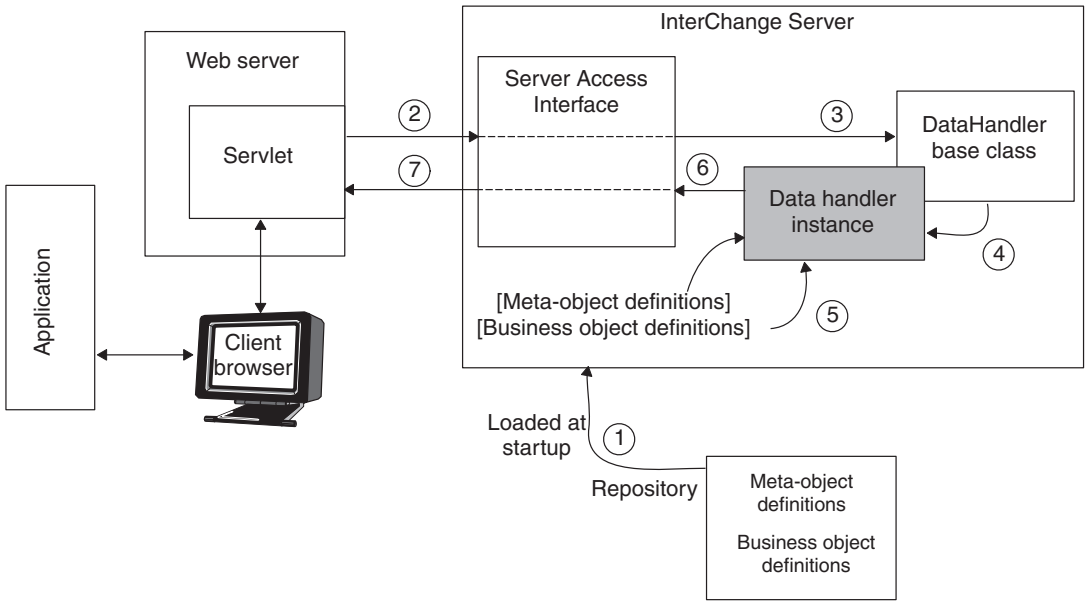

Figure 8. Data handler instantiation in the Server Access Interface context

To instantiate a data handler called in the context of the Server Access Interface, the Server Access Interface takes the following steps:

1. Meta-objects are read into memory when the server starts up, along with all other business object definitions in the repository.

   Meta-objects are stored in the repository along with business objects. The meta-object definitions, like business object definitions, must exist in memory for the data handler to be able to access them. Once these meta-objects are in memory as part of the InterChange Server (and Server Access Interface) process, the meta-objects are accessible by a data handler called in the context of the Server Access Interface.

2. An access client initiates creation of an instance of a data handler with one of the Server Access Interface methods: `IcreateBusinessObjectFrom()` or `ItoExternalForm()`.

   These methods pass the MIME type of the data to be converted.

   Note: Access clients must use Server Access Interface methods to call a data handler. Although these methods indirectly call data handler interface methods, the methods provide only a subset of the data handler interface. For information on the Server Access Interface methods, see the *Access Development Guide*. For information on the methods provided in the data handler interface, see Chapter 11, "Data Handler base class methods," on page 195.

3. The Server Access Interface sets the name of the top-level meta-object for the data handler to `MO_Server_DataHandler`.

4. The Server Access Interface creates an instance of a `DataHandler` subclass to perform the required data conversion (using the `createHandler()` method of the `DataHandler` base class).

   When called in the context of the Server Access Interface, the `createHandler()` method does *not* specify a class name. Instead, `createHandler()` converts the MIME type to a MIME-type string and obtains the name of the data handler's top-level meta-object. From this top-level meta-object, `createHandler()` obtains the name of the child meta-object for the data handler. This child meta-object

contains configuration information, including the name of the class to instantiate. For information on how this derivation occurs, see "Identifying the data-handler class" on page 13.

5. The data handler performs the required data conversion. The Server Access Interface calls the appropriate `DataHandler` method to perform the required conversion:
   - The `getBO()` method for a string-to-business-object conversion
   - The `getStringFromBO()` method for a business-object-to-string conversion or the `getStreamFromBO()` method for a business-object-to-stream conversion.
6. The data handler returns the requested format to the Server Access Interface.
7. The Server Access Interface returns the requested format to the access client.

Metadata-driven data handler design

IBM-delivered data handlers are metadata-driven. Metadata is data about the business object that is stored in business object definitions. Metadata in a business object definition provides information describing the data in an instance of the business object. In general, business object metadata includes the structure of the business object, the settings of its attribute properties, and the content of its application-specific information. It also provides instructions on how to process the data.

Connectors are typically designed to use business object metadata when processing business objects. Similarly, data handlers are designed to use business object metadata. For example:

- The XML data handler requires that each business object definition contain application-specific information describing each attribute. This text enables the data handler to identify XML elements, XML attributes, processing instructions, and other types of XML markup.
- The FixedWidth data handler uses the value of the business object attribute property MaxLength to parse fixed-width strings.
- The NameValue data handler uses the name and value of business object attributes.

A metadata-driven data handler handles each business object that it supports based on metadata encoded in the business object definition rather than on information hard-coded in the data handler. Therefore, a data handler can handle new or modified business objects without requiring modifications to the data handler code.

Chapter 2. Installing and configuring data handlers

This chapter describes how to install and configure data handlers. It also discusses how to configure a connector to support data handlers. This chapter contains the following sections:

Installing data handlers

You can install data handlers as part of either the IBM WebSphere InterChange Server or the WebSphere Business Integration Adapters product. The following sections describe the installation environment and the steps to install data handlers:

Data handlers in IBM WebSphere InterChange Server

The IBM WebSphere InterChange Server product includes the base data-handler file, `CwDataHandlers.jar`. Therefore, this product includes the data handlers that Table 3 on page 5 lists. The Server Access Interface process (within InterChange Server) can access any of the data handlers in the `CwDataHandlers.jar` file. The InterChange Server Installer automatically installs this data-handler file. For the use of the InterChange Server Installer, follow the instructions in the *System Installation Guide for UNIX* or *for Windows*.

Note: To use additional IBM-delivered data handlers with InterChange Server, you must purchase the IBM WebSphere Business Integration Adapters product. This product includes all IBM-delivered data handlers as well as the sample data-handler code to assist in the development of custom data handlers.

When the installation is complete, the files in Table 6 are installed in the product directory on your system.

Table 6. Installed file structure for data handlers in WebSphere InterChange Server

Directory	Description
`DataHandlers`	Contains the file `CwDataHandler.jar` for compiled versions of the IBM-delivered data handlers.
`repository\edk`	Contains text files with meta-object definitions for data handlers called in the context of the Server Access Interface (used with the InterChange Server integration broker).

Note: In this chapter, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes. All pathnames are relative to the product directory, which is the directory where the WebSphere business integration system is installed.

Data handlers in IBM WebSphere Business Integration Adapters

The IBM WebSphere Business Integration Adapters product includes all IBM-delivered data handlers (see "IBM-delivered data handlers" on page 4). To install the data handlers, use the WebSphere Business Integration Adapters Installer, which installs them automatically.

To install the separate XML or EDI data handlers, you must follow the instructions in "Installing data handlers using the graphical installer" or "Installing data handlers silently" on page 23.

When the installation is complete, the files in Table 7 are installed in the product directory on your system.

Table 7. Installed file structure for data handlers in WebSphere Business Integration Adapters

Directory	Description
`DataHandlers`	Contains the file `CwDataHandler.jar` for compiled versions of the IBM-delivered base data handlers. Also contains an empty jar file, `CustDataHandler.jar`, which is intended to hold any custom data handlers you might develop.
`DevelopmentKits\edk\DataHandler`	Contains a template file (`StubDataHandler.java`) for a custom data handler and a batch file (`make_datahandler.bat`)to compile a custom data handler.
`DevelopmentKits\edk\DataHandler\Samples`	Contains source code for the sample FixedWidth, NameValue, and Delimited data handlers.
`repository\DataHandlers`	Contains text files with meta-object definitions for data handlers called in the context of a connector.

Note: In this chapter, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes. All pathnames are relative to the product directory, which is the directory where the WebSphere business integration system is installed.

Installing data handlers using the graphical installer

You must do the following to install the separate XML and EDI data handlers using the graphical installer:

1. Invoke Installer as described in for your data handler.
2. At the language selection prompt, choose the desired language from the drop-down menu and click **OK**.
3. At the Welcome screen click **Next**.
4. At the IBM license acceptance screen, click **I accept the terms in the license agreement** and then click **OK**.
5. The product directory screen allows you to specify where to install the data handlers. The product directory must contain an installation of a compatible version of the adapter framework. You may specify another directory, but it must contain an installation of a compatible version of the adapter framework.

   On the Windows platform, the field defaults to the value contained in the `CROSSWORLDS` environment variable, which is set by the either the WebSphere InterChange Server Installer or the WebSphere Business Integration Adapters Installer for adapter framework.

   On the UNIX platform, the installer queries an entry in a file created by either the adapter framework installer or the WebSphere InterChange Server installer.

At the product directory screen, perform one of the following tasks:

- Type the full path of the directory into which you want to install the adapter framework in the **Directory Name** field and click **Next**
- Click **Browse** to select a directory and click **Next**
- Accept the default path and click **Next**

6. The summary screen lists the features you selected for installation, the specified product directory, and the amount of disk space required. Read the information to verify it and then click **Next**.
7. If you are installing on a Windows computer then Installer presents the program folder selection screen for some data handlers. In the **Program Group** field, type the name of the program group in which you want to create shortcuts for the adapters or accept the default program group and then click **Next**.
8. After Installer finishes successfully, click **Finish**.

Installing data handlers silently

You must do the following to perform a silent installation of the separate XML and EDI data handlers:

1. Prepare a response file to install the data handler using the desired options listed in Table 8.

Table 8. Silent installation options for data handlers

Option name	Option values
`-W installLocation.active`	
`-W installLocation.value`	Set to the directory path in which InterChange Server is installed. If you leave this option commented, the product installs to the default directory listed in . This option is only relevant for the WICS broker when installed on Windows. Make sure it is commented out when installing for the WMQI or WAS brokers, or when installing on a UNIX computer.
`-G replaceExistingResponse`	Set to `yesToAll` or `yes` to replace all files found on the system that have the same name as those being copied by the installer. Set to `noToAll` or `no` to not replace any files found on the system that have the same name as those being copied by the installer.
`-G replaceNewerResponses`	Set to `yesToAll` or `yes` to replace all files found on the system that are newer than those being copied by the installer. Set to `noToAll` or `no` to not replace any files found on the system that are newer than those being copied by the installer.
`-G createDirectoryResponse`	Set to `yes` to create the product directory specified by the option if it does not already exist. Set to `no` to not have the product directory created if it does not exist. You must set this option to `yes` if the specified directory does not exist for the installation to succeed.

Table 8. Silent installation options for data handlers (continued)

Option name	Option values
`-G removeExistingResponse`	This option is not relevant for any broker on any platform. Comment out this option.
`-G removeModifiedResponse`	This option is not relevant for any broker on any platform. Comment out this option.

2. Perform the silent installation as described in using the response file prepared in step 1 on page 23.

Configuring data handlers

Data-handler meta-objects allow a connector or Server Access Interface process (if InterChange Server if your integration broker) to instantiate a data handler based on the MIME type of an input file or the MIME type specified in a business object request. To configure a data handler, you must ensure that its meta-objects are correctly initialized and available to the calling components (a connector or an access client).

Note: Each IBM-delivered data handler uses configuration properties that are defined in data-handler meta-objects. However, a custom data handler might or might not use meta-objects for its configuration properties. For more information, see "Using data-handler meta-objects" on page 171.

In support of the IBM-delivered data handlers, IBM delivers the data-handler meta-objects listed in Table 9.

Table 9. IBM-delivered data-handler meta-objects

Meta-object level	Quantity	Location
Top-Level		
For InterChange Server	One	`repository\edk`
For connector	One	`repository\DataHandlers`
Child	One for each data handler	`repository\DataHandlers`

To configure the use of one or more data handlers for use by a caller, you must:

- In a top-level meta-object, provide the caller with the supported MIME types and their associated data handlers.
- In a child meta-object, provide the caller with the appropriate configuration information for the desired behavior of the data handler.

Top-level meta-objects

A top-level data-handler meta-object associates a MIME type with a child data-handler meta-object. The child meta-object provides configuration information, which always includes the name of the data handler class to instantiate. Therefore, a top-level meta-object associates a MIME type with a data handler. All calling components with access to a particular top-level meta-object can invoke any of the data handlers whose MIME type appears in this meta-object.

You can control which data handlers a calling component can support by grouping the appropriate MIME-type attributes in a particular top-level meta-object and

having the calling component provide the name of the meta-object that contains the data handlers it needs to use. IBM delivers the following top-level data-handler meta-objects:

- `MO_Server_DataHandler` to identify data handlers that are available to access clients that invoke data handlers.
- `MO_DataHandler_Default` to identify data handlers that are available to connectors that invoke data handlers.

MO_Server_DataHandler meta-object

The Server Access Interface process uses the `MO_Server_DataHandler` meta-object to identify the data handlers it can use. The delivered version of `MO_Server_DataHandler` is not configured to support any MIME types. It includes only a single dummy attribute. You can customize this meta-object to support any data handler installed with your InterChange Server. If you want your access clients to support a MIME type, rename the `dummy` attribute in the top-level `MO_Server_DataHandler` meta-object to the name of the supported MIME type and provide the associated child meta-object for that MIME type.

For example, to provide access clients with support for the `text_xml` MIME type, rename the `dummy` attribute to `text_xml` and provide the name of the MIME type's associated child meta-object as the attribute's type. This child meta-object configures the XML data handler. Figure 9 illustrates a `MO_Server_DataHandler` meta-object that contains one attribute, `text_xml`, which represents the `MO_DataHandler_DefaultXMLConfig` child meta-object.

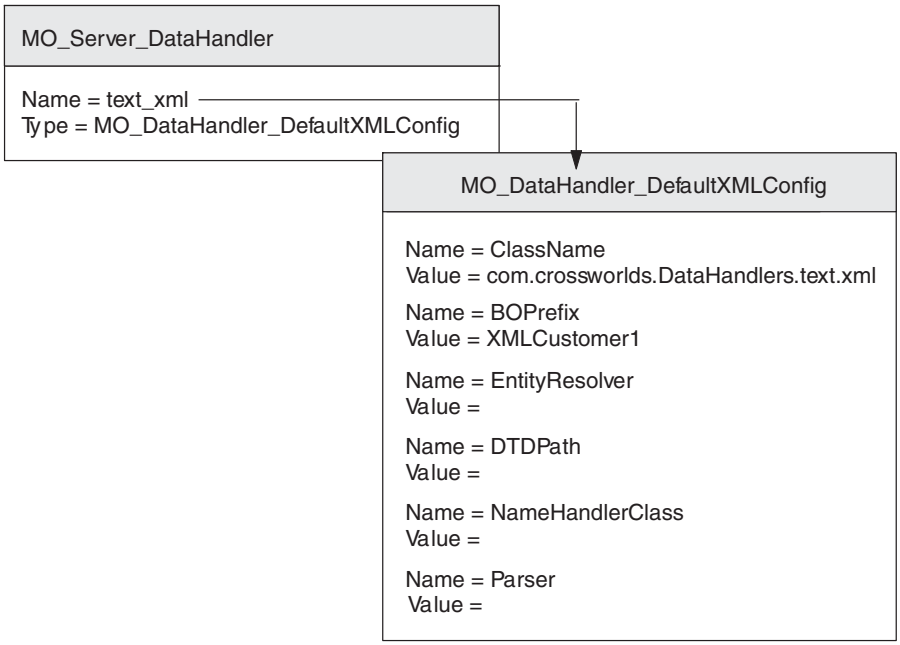

Figure 9. MO_Server_DataHandler meta-object

If you want your access clients to support additional MIME types, define a new attribute in the top-level `MO_Server_DataHandler` meta-object for each MIME type and provide the associated child meta-object for that MIME type. If you are invoking more than one data handler, you must define a child meta-object for each data handler instance. To provide support for an additional MIME type, you can either:

- Add any of the MIME types supported by the IBM-delivered data handlers (see Table 3 on page 5 and Table 4 on page 5) to the top-level data-handler meta-object.
- Define your own custom MIME type and child meta-object, as long as you have a data handler that supports it. For more information, see "Modifying the top-level meta-object" on page 188.

Note: The name of the top-level server meta-object for data handlers *must* be the default name of `MO_Server_DataHandler`, but you can configure the top-level meta-object to contain any number of child meta-objects.

MO_DataHandler_Default meta-object

By default, a connector uses the `MO_DataHandler_Default` meta-object to identify the data handlers it can use. The delivered version of `MO_DataHandler_Default` is configured to support the MIME types of all IBM-delivered data handlers (including some adapter-specific data handlers that are not covered in this document).

If you want your connector to support different MIME types, you must ensure that an attribute exists in the `MO_DataHandler_Default` meta-object for each MIME type that you want the connector to support. This attribute must specify the appropriate MIME type and represent the associated child meta-object for that MIME type. To provide support for an additional MIME type, you can define your own custom MIME type and child meta-object, as long as you have a data handler that supports it. For more information, see "Modifying the top-level meta-object" on page 188.

Note: You can change the name of the top-level meta-object for connectors to correspond to a particular connector, or even a particular business object or a particular type of file that the connector needs to process. Whatever object you use, however, must be supported by the connector definition, so if you use a different top-level object you must configure the connector definition to support it. For more information, see "Configuring a connector to use data handlers" on page 28.

Figure 10 shows a top-level data-handler meta-object for connectors that defines two data handlers: XML and NameValue.

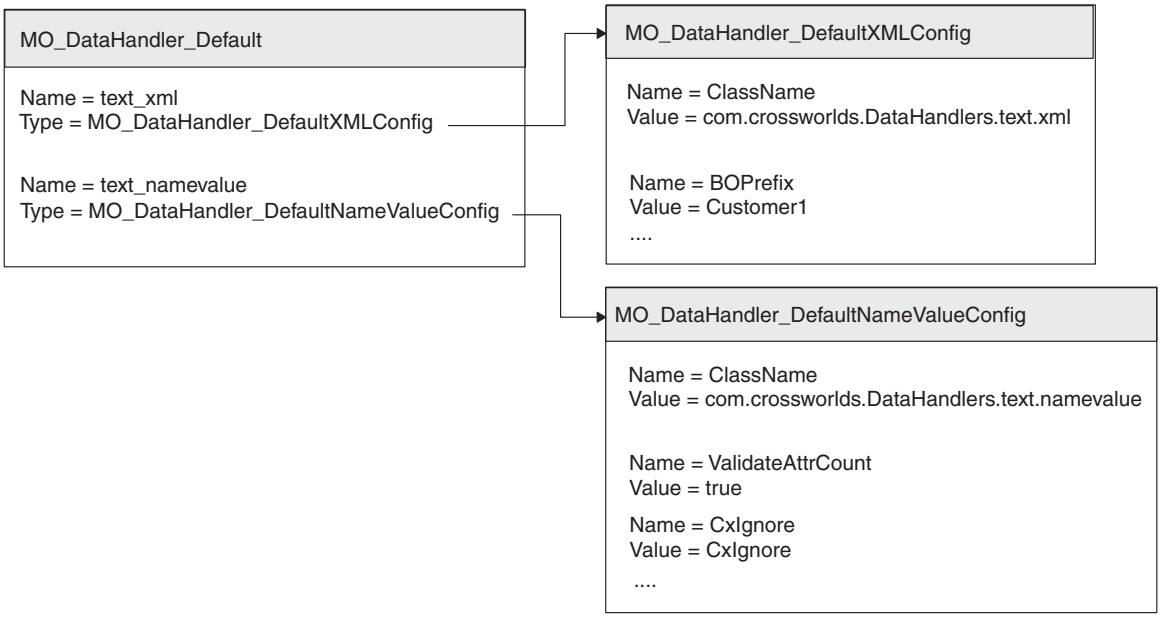

Figure 10. Example meta-object for two different data handlers

Note: For a connector to be able to access a data handler, the top-level data-handler meta-object must be on the list of supported objects for the connector. Otherwise, the connector cannot load the meta-object at startup.

Child meta-objects

A child data-handler meta-object is a flat business object that contains configuration information to initialize a data handler. Different types of data handlers have different configuration requirements, so child meta-objects have different attributes. This configuration information customizes the behavior of the data handler instance. Therefore, a set of attribute values in a child meta-object defines a specific configuration, which in turn is associated with a particular data handler behavior. All callers that access a particular child meta-object invoke the behavior of the associated data handler that the configuration information defines.

- If all callers that access a given top-level meta-object require *only one* behavior of a particular data handler, provide appropriate configuration information in a child meta-object and associate this child meta-object with data handler's MIME type in the top-level meta-object.

  For example:

  – For all connectors to access the particular data-handler behavior, make sure that the child meta-object is associated with the data handler's MIME type in the top-level meta-object for connectors (`MO_DataHandler_Default` by default).

  – InterChange Server integration broker only — For all access clients to access the particular data handler behavior, make sure that the child meta-object is associated with the data handler's MIME type in the top-level meta-object for the server, `MO_Server_DataHandler`.

- If callers that access a given top-level meta-object require *more than one* behavior of a particular data handler, create a child meta-object with the appropriate configuration information for each data-handler behavior and associate each child meta-object with a unique MIME-type name (in the top-level data-handler meta-object).

  IBM recommends that you name the child meta-objects with a unique MIME type/subtype combination:

`text_MIMEtype_subtype`

where:

– The MIME type (*MIMEtype*) represents the MIME type that the data handler supports.

– The MIME subtype (*subtype*) represents the particular behavior of the data handler.

For example, if all connectors can support *both* the default XML data handler and an SGML version, you can create the following MIME types: `text_xml` and `text_xml_sgml`. Keep in mind that MIME-type names can only contain alphanumeric characters and the special characters of period (.) and underscore (_).

IBM delivers a child data-handler meta-object for each of the data handlers it delivers, as Table 10 shows.

Table 10. Child data-handler meta-objects

Child meta-object	For more information
`MO_DataHandler_DefaultXMLConfig`	"Configuring the XML data handler" on page 39
`MO_DataHandler_DefaultEDIConfig`	"Configuring the EDI data handler" on page 90
`MO_DataHandler_DefaultFixedWidthConfig`	"Configuring the FixedWidth data handler" on page 132
`MO_DataHandler_DefaultDelimitedConfig`	"Configuring the Delimited data handler" on page 140
`MO_DataHandler_DefaultNameValueConfig`	"Configuring the NameValue data handler" on page 148
`MO_DataHandler_DefaultRequestResponseConfig`	"Configuring the Request-Response data handler" on page 123

Configuring a connector to use data handlers

If a data handler is to run in the context of a connector, you must configure the connector to use a data handler:

- A connector must have access to the data-handler meta-object in order to instantiate a data handler.

  Before invoking a data handler for the first time, a connector sets a static property in the data handler base class to the name of a top-level data-handler meta-object. From this top-level meta-object, the data handler obtains its configuration information. Each time the data handler is subsequently instantiated, the configuration properties for that data handler instance are obtained. The data handler must have access to this configuration information to do its work.

- To instantiate the data handler, the connector must know either the name for the data handler class or the MIME type of the data.

  When a connector calls `createHandler()` to invoke a data handler, it passes in either the class name or the MIME type for the data.

  – If the connector passes in the MIME type, the `createHandler()` method checks the top-level data-handler meta-object for an attribute whose name matches the MIME type. If a matching attribute is found, the `createHandler()` method checks for the value of the `ClassName` attribute in the child meta-object that is associated with the MIME type.

  – If the connector passes in a class name, the `createHandler()` method instantiates a data handler of that class name.

If the connector does not pass in the correct class name or MIME type, the instantiation process fails. For more information, see "Identifying the data-handler class" on page 13.

Connectors are configured to obtain this configuration information in different ways. For example:

- The WebSphere Business Integration Adapter for XML has a configuration property, `DataHandlerConfigMO`, that specifies the name of the top-level meta-object. If this property is not filled in, the connector cannot find the meta-object. In addition, any top-level business object for the XML connector must have a `MimeType` attribute that specifies the MIME type of the data in the business object. The connector uses the `MimeType` attribute value to invoke the appropriate data handler.
- The WebSphere Business Integration Adapter for JText has its own configuration meta-object, which has `ClassName`, `DataHandlerConfigMO` and `MimeType` attributes to specify the name of the class, data-handler meta-object, and the MIME type for a file, respectively.

Other connectors may have different ways of configuring the use of a data handler. See the adapter guide for the connector for more information.

If the connector cannot find the data handler top-level meta-object, or it cannot determine the class name or MIME type, then it cannot create the data handler. Therefore, when you are configuring a connector to use a data handler, be sure to:

1. Determine how to configure the name of the top-level data-handler meta-object for the connector. Make sure that the spelling of the meta-object name is correct.
2. Determine how to configure the MIME type. Make sure that the MIME type is spelled correctly.
3. Make sure that the top-level data-handler meta-object is in the supported objects list for the connector.
4. Make sure that the child meta-object for the data handler has the value of the data handler class name (in the `ClassName` attribute) specified correctly.

Chapter 3. XML data handler

The IBM WebSphere Business Integration Data Handler for XML, called the *XML data handler*, converts business objects into XML documents and XML documents into business objects. For instructions on installing the XML data handler, see "Installing data handlers" on page 21.

Note: The XML data handler supports XML version 1.0.

This chapter describes how the XML data handler processes XML documents and how to define business objects to be processed by the XML data handler. It also discusses how to configure the XML data handler. This chapter contains the following sections:

Overview

The XML data handler is a data-conversion module whose primary role is to convert business objects to and from XML documents. An XML document is serialized data with the `text/xml` MIME type. The XML data handler can be used by connectors and by access clients.

This overview provides the following information about the XML data handler:

Processing XML documents and business objects

XML documents use a template, called a schema, to define their structure. Table 11 shows the most common data models for defining this schema.

Table 11. XML data models

XML data model	For more information
Document type definitions (DTDs) Schema documents	"XML documents that use DTDs" on page 42 "XML documents that use schema documents" on page 54

Just as a DTD or a schema document describes the structure of the XML document, the business object definition describes the structure of the business object. The XML data handler uses business object definitions when it converts between business objects and XML documents. It determines how to perform the conversion

using the structure of the business object definition and its application-specific information. A properly-constructed business object definition ensures that the data handler can correctly convert a business object to an XML document and an XML document to a business object. Before the XML data handler can perform a conversion between XML document and business object, it must be able to locate the associated business object definition.

Use of the XML data handler to convert an XML document to a business object or a business object to an XML document requires that the following steps occur.

Table 12. Using the XML data handler

Step	For more information
1. Business object definitions that describe the XML and business-object structure must exist and be available to the XML data handler when it executes.	"Requirements for business object definitions" on page 34 "Creating business object definitions from DTDs" on page 53
2. The XML data handler must be configured for your environment.	"Configuring the XML data handler" on page 39
3. The XML data handler must be called from a connector (or access client) to perform the appropriate data operation:	
a) Data operation: receive a business object from the caller, convert the business object to an XML document, and pass the XML document to the caller.	"Converting business objects to XML documents" on page 82
b) Data operation: Receive an XML document from the caller and use the name handler and SAX parser to build a business object. Then return the business object to the caller.	"Converting XML documents to business objects" on page 85

XML data handler components

The XML data handler uses following components to convert XML data to a business object:

- Name handler
- Simple API for XML (SAX) parser
- (Optional) If the XML document has a DTD and it includes entity references, the XML data handler uses an additional component—the entity resolver—to resolve the references.

Figure 11 illustrates the XML data handler components and their relationship to one another. These components are described in the sections that follow.

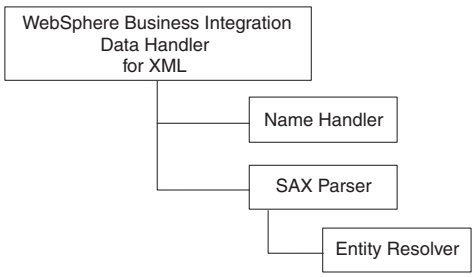

Figure 11. XML data handler components

Name handler

The XML data handler uses the name handler to extract the name of the business object from an XML message. The data handler invokes an instance of the name handler based on the value of the `NameHandlerClass` attribute in the XML data handler child meta-object:

- If a class name is provided in the `NameHandlerClass` attribute, the XML data handler uses that name handler to determine the business object name.
- If no class name is provided, the data handler uses the default name handler to determine the business object name. The default name handler uses the name of the root element in the XML document and `BOPrefix` property to form the business object name:

  BOPrefix_rootElement

For information on how to create a custom name handler, see "Building a custom XML name handler" on page 86.

SAX parser

If a parser is *not* specified in the Default Value property of the `Parser` attribute in the XML child meta-object, the data handler uses the default SAX Parser:

```
org.apache.xerces.parsers.SAXParser
```

To use a validating parser, you can take either of the following steps:

- Set the Default Value property of the `Validation` attribute in the XML child meta-object to true. The Default Value that IBM provides for this attribute is false.
- To use a validating SAX parser from IBM, change the class name in the Default Value property of the `Parser` attribute to:

  ```
  com.ibm.xml.parsers.ValidatingSaxParser
  ```

  If your business object definitions are based on DTDs, use the local entity resolver, and provide a Default Value for the document type definition (DTD) or schema path in the `DTDPath` attribute. Make sure you place all the DTD's or Schema files in the location specified in the DTDPath attribute.

Note: When using a validating parser, make sure that you are using the correct `EntityResolver` and that the `DTDPath` is set correctly. For instruction on how to do this, see "Configuring the XML data handler" on page 39.

Alternatively, you may use the non-validating SAX Parser from IBM. To use this parser, set the Default Value property of the Parser attribute of the XML child meta-object to the value `com.ibm.xml.parsers.SAXParser`.

Entity resolver

The entity resolver specifies how the SAX parser resolves external references (such as references to DTDs and schema documents) in XML data. If the XML document contains entity references, the SAX parser invokes an instance of the entity resolver using the `EntityResolver` attribute in the XML data handler configuration meta-object.

External references are handled differently depending on the entity-resolver class that `EntityResolver` specifies. Table 13 shows the entity-resolver classes that the XML data handler provides.

Table 13. Entity-resolver classes for the XML data handler

Entity-resolver class	Description
`DefaultEntityResolver`	This class is the default entity resolver. If this entity resolver is invoked, all external references are ignored.
`LocalEntityResolver`	The local entity resolver processes external references as local file names. Its behavior depends on the data model used for validation: • If DTDs are used for validation, the local entity resolver substitutes the path in the `systemID` with the value of the `DTDPath` meta-object attribute, if the `systemID` starts with `file://` or `http://` and the `DTDPath` attribute is set. The external reference is ignored if the `systemID` is not a path name or the `DTDPath` attribute is not set. • If schema documents are used for validation, the local entity resolver substitutes the path that the `schemaLocation` or `noNamespaceSchemaLocation` attribute specifies with the value of the `DTDPath` meta-object attribute, if path starts with `file://` or `http://`, or it contains a DOS filename (for example, `"D:\xmlschemas\test"`).
`URIEntityResolver`	This entity resolver processes external references as local file names or downloadable URLs. It dynamically resolves the external reference in either of the following cases: • If DTDs are used for validation: if the `DOCTYPE` contains a `SYSTEM` value that begins with `http://` or file:// • If schema documents are used for validation: if the `schemaLocation` or `noNamespaceSchemaLocation` attribute begins with `http://` or `file://` The entity resolver then opens an HTTP connection and downloads the DTD or schema document from the specified web site. **Attention:** The XML data handler does *not* cache the DTDs or schema documents. When the data handler uses the `URIEntityResolver` class as its entity resolver, it opens an HTTP connection *each time* it parses the XML document. Therefore, network traffic can impact the performance of the XML data handler.

Note: All entity-resolver classes in Table 13 must have the following class prefix:

```
com.crossworlds.DataHandlers.xml
```

If your XML documents use schema documents, any external schemas that the schema document includes are also treated as external entities. Therefore, the SAX parser invokes an entity resolver to resolve these included schema documents. If the XML document uses the `schemaLocation` or `noNamespaceSchemaLocation` to specify schema locations, you can set the `EntityResolver` attribute to either `LocalEntityResolver` or `URIEntityResolver` for validation of external schema documents (either included or imported).

If you need to specify another way to find external entities, you must create a custom entity resolver. For information on creating a custom entity resolver, see "Building a custom entity resolver" on page 88.

Requirements for business object definitions

To ensure that business object definitions conform to the requirements of the XML data handler, use the guidelines in this section, which involve:

- "Business object structure" on page 35
- "Business object attribute properties" on page 36

A properly-constructed business object definition ensures that the data handler can correctly convert a business object to an XML document and an XML document to a business object. For information on how to create business objects for the XML data handler, see "Creating business object definitions from DTDs" on page 53.

Business object structure

To represent a DTD or schema document requires at least two business object definitions:

- The *top-level* business object represents the information that defines the DTD or schema document and must contain the following:
  - An attribute named `XMLDeclaration` to represent the XML version

    This attribute must have the `type=pi` tag in its application-specific information.
  - An attribute to represent the root element in the DTD or schema document

    This attribute must have as its type a single-cardinality business object, whose type is the business object definition for the root element of the DTD or schema document. The XML ODA obtains the name of this root element from the `Root` ODA configuration property. The application-specific information must list the name of this element with the `elem_name` tag.

    Note: The `elem_name` tag replaces previous syntax, which required only the name of the XML element in the business-object attribute's application-specific information. The XML data handler still supports the old syntax for backward compatibility with existing business object definitions. However, the XML ODA uses the new syntax when it generates business object definitions.
- A *root-element* business object definition represents the XML-definition document's root element. You can tell the XML ODA which element to consider the root element through its `Root` configuration property. It contains an attribute for each of the XML components in the root element.

A business object that is processed by the XML data handler using business object definitions from DTDs or schema documents must follow these additional rules:

- Every tag in the XML document must have an associated attribute in the business object. The business object definition provides the type of the business object attribute and the application-specific information for that attribute are stored. This information is determined by the structure and content of the XML element.
- In the business object definition for an XML element, all attributes that represent XML attributes must occur *before* other attributes. The XML data handler assumes that attributes for a given XML element are the *first* attributes in the business object definition.

Note: A business object must contain enough data so that the XML data handler can create a valid XML document. Avoid sending the data handler business objects without data.

This document provides the following information about the structure of business object definitions for DTDs and schema documents:

Data model	For more information
Document type definition (DTD)	"Business object structure for DTDs" on page 43
Schema document	"Business object structure for schema documents" on page 55

Business object attribute properties

Business object definitions define attributes. Each attribute has various properties that provide information about the attribute. This section describes how the XML data handler interprets several of these properties and describes how to set them when modifying a business object definition.

Name attribute property

Each business object attribute must have a unique name. The XML Element or XML Attribtue name is always specified in the elem_name or attr_name tag. In this case, the name of the XML element (or attribute) specified in the elem_name (or attr_name) tag of the attribute application-specific information contains the special characters. However, the name of the business object attribute (which does not allow these special characters) omits them.

Type attribute property

Each business object attribute must have a type, such as Integer, String, or the type of a contained child business object, as follows:

- For a DTD: XML elements that contain either child elements or one or more non-FIXED attributes are treated as business objects. XML elements with only a PCDATA value are treated as attributes if the XML element is included in its parent through single cardinality. If it is included through multiple cardinality, it is represented as a business object because business object definitions do not support multiple-cardinality scalar values (for example, an array of String values).
- For a schema document: Each business object attribute must have a type of either String or the type of a contained child business object. XML elements that contain either child elements or complex types are treated as business objects. XML elements with only a simple-type value are treated as business object attributes if the XML element is included in its parent through single cardinality. If it is included through multiple cardinality, it is represented as a business object because business object definitions do not support multiple-cardinality scalar values (for example, an array of String values).

Note: All simple attributes should be of type `String`

Key and Foreign Key attribute properties

Each business object must have at least one primary key attribute, which is specified by setting the Key property to true for an attribute. The setting of the Foreign Key property is optional and depends on the structure of the XML document. This section provides the following information about the Key and Foreign Key attribute properties:

- "Designating the key in the business object definition"
- "Handling keys and ″required-ness″" on page 37

Designating the key in the business object definition: In earlier versions of XML-business-object-definition-generation tools (such as XMLBorgen, Edifecs SpecBuilder, and the XML ODA), the generation tool designated the ObjectEventId

attribute as the key of a parent XML business object. However, as of this release, Business Object Designer no longer allows you to save a business object definition that has the ObjectEventId attribute specified as a key.

Because of this restriction, the current version of XML ODA now takes the following actions:

- In each child business object, it sets the first attribute as the key.
- In the parent business object, it does *not* set a key attribute.

To provide a key to a parent business object definition that the XML ODA generates, you must bring up the business object definition in Business Object Designer and analyze your business object definition to determine the appropriate attribute to designate as the key. You must change the business object definition's key attribute *before* you can save the business object definition in Business Object Designer .

Note: The XML ODA replaces earlier XML-business-object-definition-generation tools (such as XMLBorgen and Edifecs SpecBuilder). Therefore, only the XML ODA takes these special steps to avoid assignment of the ObjectEventId as the parent business object's key attribute. If you have existing XML business object definitions that you have generated with any earlier XML-business-object-definition-generation tools (including an earlier version of the XML ODA), these business object definitions might still use ObjectEventId as a key. You should analyze these business object definitions if you are migrating their business objects to the current release. Failure to set an appropriate key attribute in your business object definition can have a negative impact on the performance of the event sequencing feature.

Handling keys and "required-ness": This document provides the following information about the relationship between keys and "required-ness":

Data model	For more information
Document type definition (DTD)	"Business object attribute properties for DTDs" on page 44
Schema document	"Business object attribute properties for schema documents" on page 63

Required attribute property

If this property is specified for an attribute that contains a single-cardinality child business object, the XML data handler requires that the parent business object contain a child business object for this attribute. The settings of the Cardinality, Key, and Foreign Key attribute properties can affect the setting an attribute's Required property.

This document provides the following information about "required-ness":

Data model	For more information
Document type definition (DTD)	"Business object attribute properties for DTDs" on page 44
Schema document	"Business object attribute properties for schema documents" on page 63

Cardinality attribute property

The Cardinality property indicates the number of child business objects allowed in an attribute that has a business object definition as its type. The setting of this property depends on the structure of the XML document and its elements. Its setting also affects whether the attribute must be required (its Required property set to true).

This document provides the following information about the relationship between cardinality and ″required-ness″:

Data model	For more information
Document type definition (DTD) Schema document	“Business object attribute properties for DTDs” on page 44

Special attribute values

A business object attribute has a value whose type matches the attribute's Type property. In addition, an attribute can have either of two special values:

- CxIgnore

  When the XML data handler receives a business object from an integration broker, it ignores *all* attributes with a value of CxIgnore. It is as if those attributes are invisible to the data handler. Therefore, the data handler does *not* generate a corresponding XML element; that is, it does *not* create an XML tag for this attribute (not even an empty tag). When the XML data handler receives XML input that has no XML tag that corresponds to a business object attribute, the data handler assigns the attribute a value of CxIgnore.
- CxBlank

  When the XML data handler receives a business object from an integration broker, it processes the CxBlank attribute value based on the attribute's Type property:
  - For a complex attribute (one whose Type property is set to the name of another business object definition), the data handler assumes that no complex attributes have the CxBlank value.
  - For a simple attribute (one whose Type property is set to a `String` data type), the data handler creates an empty tag in the XML document. For XML documents based on DTDs, empty double quotation marks (″ ″) are used as the PCDATA equivalent of CxBlank.

When the XML data handler receives XML input that has an empty tag, the data handler assigns a value of CxBlank to its corresponding business object attribute.

Application-specific information

Application-specific information in business object definitions provides the data handler with instructions on how to convert business objects to XML documents. The application-specific information enables the data handler to process the business object correctly. Therefore, if you create new business objects or modify existing business objects for the XML data handler, make sure that the application-specific information in the business object definition matches the syntax that the data handler expects. The XML data handler can use the following kinds of application-specific information:

- Business-object-level application-specific information provides information about the business object definition as a whole.

- Attribute-level application-specific information provides information about a particular attribute.

Note: The XML data handler uses application-specific information to match components of an XML document with attributes in a business object. The maximum length for application-specific information is 255 characters. If the value of the application-specific information is more than 255 characters, you must reconstruct your DTD or schema document, and then regenerate the business object.

This document provides the following information about application-specific information:

Data model	For more information
Document type definition (DTD)	"Application-specific information for XML components in DTDs" on page 45
Schema document	"Application-specific information for XML components in schema documents" on page 64

Business object verbs

When converting business objects to XML documents, the XML data handler does *not* generate XML for the verb, nor does it set a verb when converting an XML document to a business object. However, verb information can be preserved in one of these ways:

- You can create an element in the DTD or schema document for the verb and create a business object attribute for the verb. You can then design the content of your business integration system to copy the verb into the business object attribute. The data handler then converts the verb to the XML element, thereby preserving the verb in the XML document. When the XML document is returned, the business integration system can set the verb according to the value of the business object attribute.
- You can create DTDs or schema documents for specific business object and verb combinations, and associate each business object request with the DTD or schema document for that business object and verb. When an XML document is converted to a business object and returned to the caller, the connector can set the verb that corresponds to the DTD or schema document.
- If the calling connector can be provided with information about the verb, it can set the verb in the business object before sending it to integration broker.

Configuring the XML data handler

To configure an XML data handler, you must ensure that its configuration information is provided in the XML data handler's child meta-object.

Note: To configure an XML data handler, you must also create or modify business object definitions so that they support the data handler. For more information, see "Requirements for business object definitions" on page 34.

For the XML data handler, IBM delivers the default child meta-object `MO_DataHandler_DefaultXMLConfig`. Each attribute in this meta-object defines a configuration property for the XML data handler. Table 14 describes the attributes in this child meta-object.

Table 14. Child meta-object attributes for the XML data handler

Attribute name	Description	Delivered default value
BOPrefix	Prefix used by the default NameHandler class to build the names of business object names. The default value must be changed to match the name of the associated the business object definition. The attribute value is case-sensitive.	XMLTEST
ClassName	Name of the data-handler class to load for use with the specified MIME type. The top-level data-handler meta-object has an attribute whose name matches the specified MIME type and whose type is the XML child meta-object (described by Table 14).	com.crossworlds. DataHandlers. text.xml
DefaultEscapeBehavior	If an attribute value contains special characters, it requires the XML data handler to perform escape processing. If the attribute's application-specific information does *not* include the escape tag, the XML data handler checks the value of the DefaultEscapeBehavior property to determine whether to perform escape processing. For more information, see "For an XML element or attribute that contains special characters" on page 51.	true
DTDPath	Used by the data handler to configure the path to the document type definitions (DTDs) or schemas (XSDs).	*None*
DummyKey	Key attribute; not used by the data handler but required by the business integration system.	1
EntityResolver	Name of the class to use to handle references to external entities such as a DTD or schema. For more information on values for this attribute, see "Entity resolver" on page 33.	*None*
IgnoreUndefinedAttributes	When this attribute is set to false, the XML data handler validates all XML attributes against the business object definition; it throws an exception when it encounters an undefined attribute. When this attribute is set to true, the XML data handler ignores all undefined XML attributes, generating a warning.	true
IgnoreUndefinedElements	When this attribute is set to false, the XML data handler validates all XML elements against the business object definition; it throws an exception when it encounters an element not defined in the application-specific information. When this attribute is set to true, the XML data handler ignores all undefined XML elements (and any attributes within these undefined elements), generating a warning.	false
InitialBufferSize	Defines the initial size of the buffer that is used when converting business objects to XML. Set this value to the size, in bytes, of your XML business objects. Setting this value to a high number will speed the conversion of business objects to serialized XML.	2 MB (2,097,152 KB)
NameHandlerClass	Name of the class to use to determine the name of a business object from the content of an XML document. Change the default value of this attribute if you create your own custom name handler. For more information, see "Building a custom XML name handler" on page 86.	com. crossworlds. DataHandlers.xml. TopElementNameHandler
Parser	Package name of a SAX-compliant parser for the XML document.	*None*

Table 14. Child meta-object attributes for the XML data handler (continued)

Attribute name	Description	Delivered default value
UseNewLine	Set to true if you want each tag in the output XML to be on a new line. (The XML data handler adds extra content in the form of line feeds and carriage returns to the XML document.) Set to false if you do not want to alter the XML output.	false
Validation	This value is used by the data handler to specify that the validating parser is used. The data handler does this by setting the feature http://xml.org/sax/features/validation of the xerces parser to true. To use a validating parser, the default value must be changed to true. When Validation is set to true, the XML parser will validate the XML document against: • A DTD, if a DTD is present • A schema document, if one is specified. In this case, the XML parser does full schema checking. • Both DTDs and schema documents, if both are specified	false
ObjectEventId	Placeholder not used by the data handler but required by the business integration system.	*None*

The "Delivered default value" column in Table 14 lists the value in the Default Value property for the corresponding attribute in the delivered business object. You must examine your environment and set the Default Value properties of those attributes to the appropriate values. You must make sure that at least the ClassName and BOPrefix attributes have default values.

Note: Use Business Object Designer to modify business object definitions.

A single child meta-object can be used by different MIME type/subtype combinations if each of the combinations uses the same XML data handler configuration. If your connector requires different XML data handler configurations for different MIME types, then you must create a separate child meta-object for each data handler instance. To prepare multiple configurations of the XML data handler, take the following steps:

- Copy and rename the default XML child meta-object. A recommended approach to naming a new child meta-object is to provide subtypes to the MIME type. For example, to support both the default XML data handler and an SGML version, you can copy the default XML child meta-object and name this copy MO_DataHandler_DefaultSGMLConfig.
- Set the default values of the attributes in each XML child meta-object to configure the data handler instance.

  Create attributes in the top-level data-handler meta-object for each MIME type/subtype combination. For example, in support of the XML and SGML, you can create the following MIME types: text_xml and text_xml_sgml. Each of these attributes would represent its associated child meta-object.

  You can also configure the XML data handler to support multiple instances of the same data handler. In this case, you can create another top-level attribute named text_xml_*subtype*, where *subtype* can be an application name, as in text_xml_AppA, an application entity name, or another appropriate name.

For more information about how to configure a data handler, see “Configuring data handlers” on page 24.

Figure 12 shows an example of a top-level data-handler meta-object and its corresponding child meta-objects. Notice that there are four attributes in the top-level meta-object `MO_DataHandler_XMLSample`, but only three child meta-objects. This is because the attribute `Application_xml_AppC` uses the same child meta-object as the attribute `text_xml_AppB` to invoke the appropriate data handler.

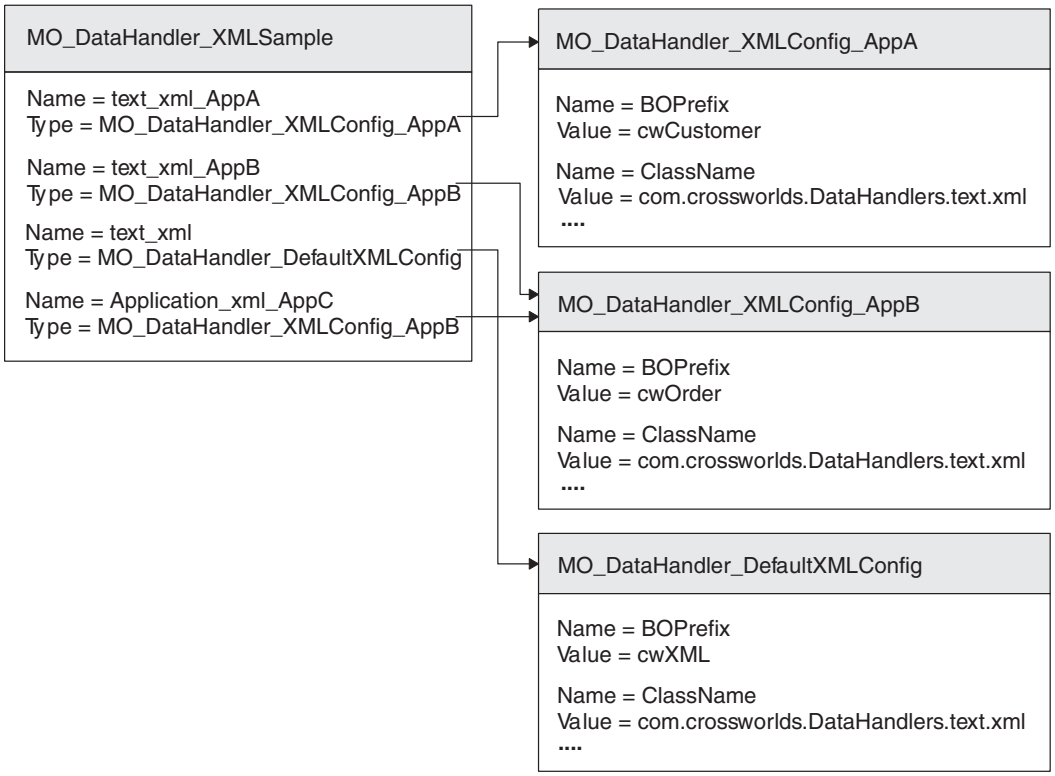

Figure 12. Example meta-object for multiple XML data handlers

XML documents that use DTDs

A *document type document* (DTD) is a data model for XML documents that provides a special syntax to describe the XML document's template, called a schema. This DTD is a file with the `.dtd` extension. The business object definitions that represent the schema of an XML document use information in the DTD to preserve and record the document's structure. This section provides the following information about deriving structure information for a business object definition from a DTD:

- “Requirements for business object definitions based on DTDs”
- “Creating business object definitions from DTDs” on page 53

Requirements for business object definitions based on DTDs

To ensure that business object definitions that correspond to DTDs conform to the requirements of the XML data handler, use the guidelines in this section, which involve:

- “Business object structure for DTDs” on page 43
- “Business object attribute properties for DTDs” on page 44
- “Application-specific information for XML components in DTDs” on page 45

A properly-constructed business object definition ensures that the data handler can correctly convert a business object to an XML document and an XML document to a business object. For information on how to create business objects for the XML data handler, see "XML documents that use schema documents" on page 54.

Business object structure for DTDs

To represent a DTD requires at least the two business object definitions described in "Business object structure" on page 35. For a DTD, these business object definitions have the following additional requirements:

- The *top-level* business object represents an XML DTD and can contain the following:
  - An attribute named `DocType` to represent the `DOCTYPE` declaration

    Whether the XML ODA generates a `DocType` attribute in the top-level business object definition depends on the setting of its `DocTypeOrSchemaLocation` configuration property. For more information, see "For an XML DOCTYPE declaration" on page 52 and "Supported DTD structures" on page 54.
  - An attribute named `XMLDeclaration` to represent the XML version

    This attribute must has the `type=pi` tag in its application-specific information. For more information, see "For XML processing instructions" on page 76.
  - An attribute to represent the root element in the DTD

    As described in "Business object structure" on page 35, this attribute must have as its type a single-cardinality business object, whose type is the business object definition for the root element. The application-specific information must list the name of this element with the `elem_name` tag. For more information, see "For XML elements" on page 72.
- A *root-element* business object definition represents the DTD's root element.

A business object that is processed by the XML data handler using business object definitions based on DTDs must also follow these rules:

- Every tag in the XML document has an associated attribute in the business object definition. The exception to this rule is for FIXED attributes. By default, FIXED attributes are *not* included in a business object definition because these attributes contain static data. However, if you want your FIXED attributes to be included in the business object definition, you can manually add attributes for them to the business object definition.

Note: For a list of general business object requirements, see "Requirements for business object definitions" on page 34.

An example DTD for an XML document is shown below. The DTD is named `Order`, and it contains elements that correspond to an application `Order` entity.

```
<!--Order -->
<!-- Element Declarations -->
<!ELEMENT Order (Unit+)>
<!ELEMENT Unit (PartNumber?, Quantity, Price, Accessory*)>
<!ELEMENT PartNumber (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ELEMENT Accessory (Quantity, Type)>
<!ATTLIST Accessory
     Name CDATA >
<!ELEMENT Type (#PCDATA)>
```

Figure 13 shows the structure of a business object that might be created to correspond to an XML document associated with the `Order` DTD. Note that each

XML element and element attribute in the Order DTD has a corresponding business object attribute. The top-level business object contains attributes for the XML declaration, the DOCTYPE, and the top-level Order element. Note also that the element attribute Name is the first attribute in the Accessory business object.

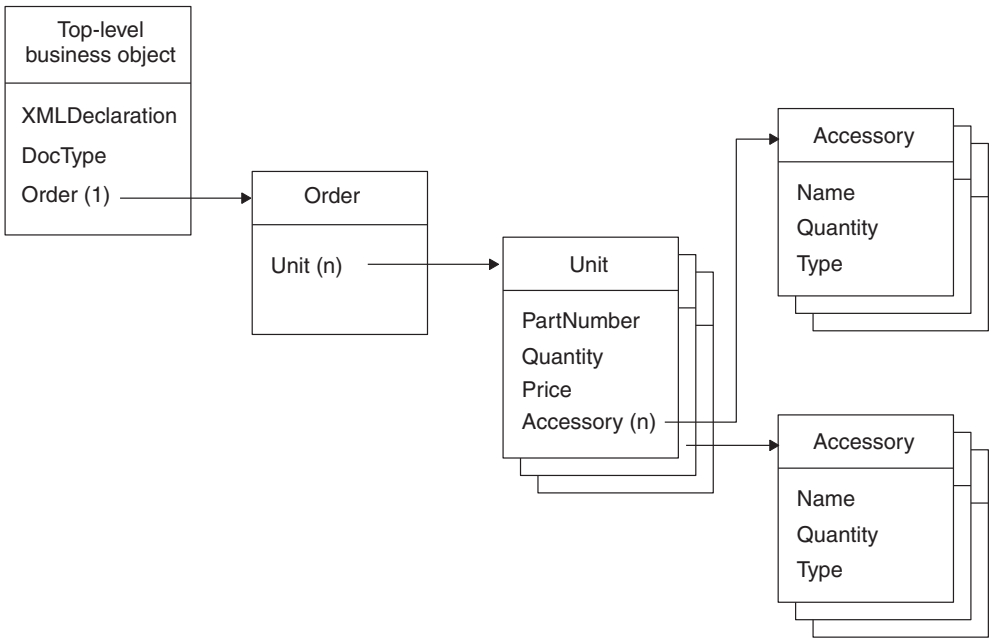

Figure 13. Example business object for XML document using the Order DTD

Business object attribute properties for DTDs

When the business object definitions for an XML document are based on DTDs, the business object attribute properties have the restrictions discussed in "Business object attribute properties" on page 36. In addition, the DTD syntax can determine the "required-ness" of a business object attribute. The "required-ness" is a combination of factors, including cardinality and whether the attribute is a key, that determines whether the XML data handler requires the attribute. If an attribute is required, its Required attribute property must be set to true.

The setting of the Required attribute property depends on the XML element and attribute specifications, as well as the settings of the Cardinality, Key, and Foreign Key attribute properties, as follows:

- The cardinality of a business object attribute is determined by the ELEMENT fragment in the DTD. This cardinality affects whether the attribute is required. Table 15 outlines the cardinality and "required-ness" for possible combinations of element declarations in a DTD.

Table 15. Cardinality and "Required-ness" for a DTD

DTD ELEMENT fragment	Cardinality	Required
None specified	1	Yes
?	1	No
+	N	Yes
*	N	No

- Whether a business object attribute is a primary or foreign key is determined by the ATTLIST fragment in the DTD. The presence of a key affects whether the attribute is required. Table 16 outlines how syntax in the ATTLIST fragment

affects the business object attribute's ″required-ness″ for possible combinations of attribute declarations in a DTD.

Table 16. Keys and ″Required-ness″ for a DTD

DTD ATTLIST fragment	Key	Required	Comment
#IMPLIED	No	No	
#REQUIRED	No	Yes	
ID {#IMPLIED | #REQUIRED}	Yes	No	#IMPLIED, #REQUIRED ignored
IDREF {#IMPLIED, #REQUIRED}	Foreign key	Depends on whether #IMPLIED or #REQUIRED is specified	
NMTOKEN {#IMPLIED | #REQUIRED}	Yes	No	#IMPLIED, #REQUIRED ignored

Application-specific information for XML components in DTDs

This section provides the following information on the application-specific information format for business object definitions based on DTDs:

- "Business-object-level application-specific information"
- "Array attribute application-specific information" on page 48
- "Attribute application-specific information" on page 48

Business-object-level application-specific information: The XML data handler uses the following types of business objects to represent different kinds of XML elements generated from a DTD:

- "Regular business object definitions based on DTDs"
- "Mixed business object definitions based on DTDs"
- "Wrapper business object definitions based on DTDs" on page 46

These types of business objects are distinguished by the application-specific information at the business object level.

Regular business object definitions based on DTDs: A *regular* business object represents an XML element. In this type of business object, the application-specific information at the business object level identifies the name of the XML element that the business object represents. For example, suppose the XML element is defined as:

```
<!ELEMENT Unit(...)>
```

The application-specific information at the business object level for the associated business object definition is:

```
[BusinessObjectDefinition]
Name = MyApp_Unit
AppSpecificInfo = elem_name=Unit
[Attribute]
...
```

Mixed business object definitions based on DTDs: A *mixed* business object represents a mixed XML element, one that contains mixed content of character data (#PCDATA) and other subelements. The DTD representation of a mixed-type XML element looks like the following:

```
<!ELEMENT (#PCDATA | CONTAINED_ELEMENT1 | CONTAINED_ELEMENTN)*>
```

For example, suppose the Cust XML element is defined in the DTD as follows:

```
<!ELEMENT Cust(#PCDATA | Address | Phone)*>
```

To represent a mixed-type XML element, use a mixed-type business object definition. For a mixed business object definition, its business-object-level application-specific information consists of the following components:

- The name of the mixed XML element
- The tag type=MIXED

For the business object definition, MyApp_Cust, which represents the Cust element, the application-specific information at the business object level is as follows:

```
[BusinessObjectDefinition]
Name = MyApp_Cust
AppSpecificInfo = Cust;type=MIXED;
```

Wrapper business object definitions based on DTDs: A wrapper business object represents a repeating choice list. This type of business object definition is needed when an XML element has children that can appear in any order and be of any cardinality. A wrapper business object preserves the order and cardinality of the child elements in the XML document.

For a choice-list XML element, the DTD definition looks like this:

```
(CONTAINEDELEMENT1 | ... | CONTAINEDELEMENTN)*
```

As an example, the choice-list XML element definition in a DTD might be:

```
<!ELEMENT CUST( U | I | B )* )>
```

This element contains three subelements that are optional and that can appear in any order. Each subelement is a simple element. Figure 14 shows an XML document of this type.

```
<CUST>
   <U>.....
   </U>
   <B>.....
   </B>
   <I>.....
   </I>
   <B>.....
   </B>
   <U>.....
   </U>
...
```

Figure 14. XML document content for a repeating choice list

To represent a choice-list XML element defined in a DTD, the business object definition is hierarchical. It includes the following business object definitions:

- The parent business object definition

  This parent business object definition contains a single attribute that represents a multiple-cardinality business object array. This attribute has as its type the business object definition for the associated wrapper business object. The parent business object definition contains the following application-specific information:

  – At the business-object-level, the parent business object definition contains the name of the choice-list element in its application-specific information

– At the attribute level, the multiple-cardinality attribute (in the parent business object definition) specifies the optional choice elements in the following format:

  (*choiceElement1*|...|*choiceElementN*)

  where *choiceElement1*...*choiceElementN* correspond to each of the choice elements defined. The pipe (|) character must separate each of the choice elements and the entire tag must be enclosed in parentheses.

- A wrapper business object definition

  The wrapper business object definition contains one attribute for each of the choice elements defined in the choice-list element. It does *not* require any application-specific information at the business-object level.

Figure 15 shows an illustration of the hierarchy of the business object definitions for a choice-list XML element. At runtime, each child business object is an instance of a wrapper business object and has only one attribute populated with data. As an example, a business object for the XML content in Figure 14 on page 46 would have five children, each with the appropriate attribute populated.

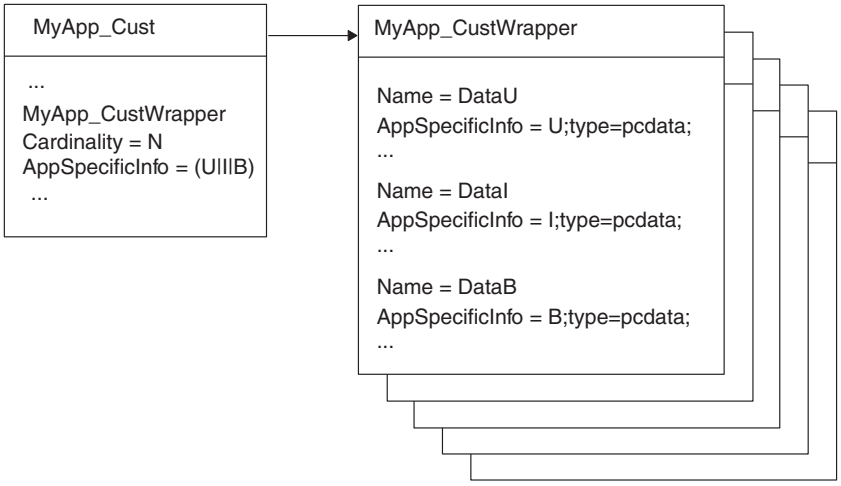

Figure 15. Hierarchical business object definition for choice-list XML element

Figure 16 shows the parent business object definition, `MyApp_Cust`, with its application-specific information.

```
[BusinessObjectDefinition]
Name = MyApp_Cust
AppSpecificInfo =

   [Attribute]
   Name = CustWrapper
   Type = MyApp_CustWrapper
   Cardinality = N
   AppSpecificInfo = attr_name=CustWrapper;(U|I|B)
[End]
```

Figure 16. Parent business object definition for a choice-list element

The wrapper business object definition, `MyApp_CustWrapper`, has three attributes, one for each choice element. Because each choice element contains character data, the application-specific information for each attribute specifies:

- The name of the element

- The tag `type=pcdata`

Note: For more information on attributes for character data, see "For an XML element with only PCDATA" on page 50.

Figure 17 shows the wrapper business object definition for this XML document.

```
[BusinessObjectDefinition]
Name = MyApp_CustWrapper
AppSpecificInfo =

   [Attribute]
   Name = DataU
   Type = String
   AppSpecificInfo = attr_name=U;type=pcdata;
   [End]

   [Attribute]
   Name = DataI
   Type = String
   AppSpecificInfo = attr_name=I;type=pcdata;
   [End]

   [Attribute]
   Name = DataB
   Type = String
   AppSpecificInfo = attr_name=B;type=pcdata;
   [End]
```

Figure 17. Wrapper business object definition for a choice-list element

Array attribute application-specific information: If a business object attribute represents an XML element that contains other elements, the application-specific information must contain the name of the element. For example, if an attribute named `DeliveryDate` has a business object type and represents an element named DATETIME, the application-specific information contains the name of the element:

```
Name = DeliveryDate
Relationship = Containment
Cardinality = n
AppSpecificInfo = DATETIME
```

Attribute application-specific information: The attribute of a business object definition can represent the following XML components:

- "For XML elements" on page 49
- "For an XML element with only PCDATA" on page 50
- "For an XML attribute" on page 50
- "For an XML element with character data and attributes" on page 51
- "For an XML element or attribute that contains special characters" on page 51
- "For an XML DOCTYPE declaration" on page 52
- "For a CDATA section" on page 52
- "For an XML comment" on page 53
- "For XML processing instructions" on page 53

Table 27 shows the tags for attribute-level application-specific information for these different XML components along with the sections in this manual that describe these tags in more detail.

Table 17. Tags for attribute application-specific information

Representation of business object attribute	Application-specific information	For more information
An XML element	`elem_name=`*`name of XML element`*	"For XML elements"
An XML element with only PCDATA	`elem_name=`*`name of XML element`*`;type=pcdata`	"For an XML element with only PCDATA" on page 50
An attribute for an XML element	`attr_name=`*`name of XML attribute`* `type=attribute`	"For an XML attribute" on page 50
An XML element that contains character data and attributes	`type=pcdata;notag`	"For an XML element with character data and attributes" on page 51
An XML element or attribute whose content includes special characters	`escape=true`	"For an XML element or attribute that contains special characters" on page 76
For a DOCTYPE declaration	`type=doctype`	"For an XML DOCTYPE declaration" on page 52
For a CDATA section	`type=cdata`	"For a CDATA section" on page 52
A comment to be added to the XML document	`type=comment`	"For an XML comment" on page 76
A processing instruction	`type=pi`	"For XML processing instructions" on page 76

Note: Attribute application-specific information can also include a tag of the form `( a | b | c )` to specify a multiple-cardinality attribute that represents a repeating choice. For more information, see "Wrapper business object definitions based on DTDs" on page 46.

For XML elements: Every simple (`String`) business object attribute that represents an XML element must include the `elem_name` tag in its application-specific information to identify the associated element:

`elem_name=`*`name of XML element`*

For example, if a business object attribute `CustLName` represents a simple XML attribute, its application-specific information is:

```
Name = CustLName
AppSpecificInfo = elem_name=CustLName;
```

XML element names can contain special characters (such as periods and hyphens). However, the names of business object attributes cannot contain these special characters. Therefore, the name of the XML element must be specified in the `elem_name` tag. To name the business object attribute, the XML ODA removes any special characters in the XML element's name, replacing them with an underscore (_) character.

In the following example, the application-specific information for the XML element specifies a different from the actual XML element's name because the attribute contains a special character:

```
Name = Phone_Tag
AppSpecificInfo = elem_name=Phone#Tag;
```

The actual name of the XML element contains a pound sign (#), which is invalid in the names of business-object attributes. Therefore, the `elem_name` tag in the

application-specific information specifies the actual XML element name. In the name of the associated business-object attribute, the pound sign is replaced with an underscore.

For an XML element with only PCDATA: XML elements that contain only character data are mixed elements, which contain only the `PCDATA` element content specifier. A business object attribute that represents an XML element with only `PCDATA` must have the following `type` tag in the application-specific information:

```
type=pcdata
```

In this case, the element name is the first field in the application-specific information, and the `type` parameter is the second field.

For example, an element named `PartNumber` that contains only `PCDATA` would have the following definition in the DTD:

```
<!ELEMENT PartNumber (#PCDATA)>
```

The corresponding attribute in the business object definition would have the following application-specific information:

```
Name = PartNumber
AppSpecificInfo = elem_name=PartNumber;type=pcdata;
```

If the application-specific information also contains the text `notag`, the XML data handler does not generate XML markup. It adds only the value of the attribute itself to the XML document. For more information, see "For an XML element with character data and attributes" on page 51.

For an XML attribute: If a business object attribute represents an attribute of an XML element, its application-specific information must include the following tags:

- The `attr_name` tag:

  ```
  attr_name=attrName
  ```

  XML attribute names can contain special characters (such as periods and hyphens). However, the names of business object attributes cannot contain these special characters. Therefore, the name of the XML attribute must be specified in the `attr_name` tag. To name the business object attribute, the XML ODA removes any special characters in the XML attribute's name.
- The `type` tag:

  ```
  type=attribute
  ```

  This `type` tag identifies the purpose of the associated business object attribute as an XML attribute.

Note: As described in "Business object structure" on page 35, all business object attributes that represent XML attributes must occur within the business object definition *before* any business object attributes that represent XML elements.

For example, if a business object attribute named `ID` represents an XML attribute named `ID`, its application-specific information is:

```
Name = ID
AppSpecificInfo = attr_name=ID;type=attribute;
```

For another example that uses the `type=attribute` tag, see "For an XML element with character data and attributes" on page 51.

For an XML element with character data and attributes: If an XML element contains *only* PCDATA or CDATA and has one or more XML attributes, its business object definition must include the following attributes:

- A business-object attribute for each XML attribute.

  The attribute name must match the name of the XML attribute. Its attribute-level application-specific information must include the attr_name and type=attribute tags. For more information, see "For an XML attribute" on page 50.
- A business-object attribute for the character data associated with the PCDATA or CDATA element-content specifier.

  This attribute contains the data that is associated with the parent XML element. Its application-specific information must contain:

  – The appropriate type=*typename* tag, (where *typename* is either pcdata or cdata) followed by a semicolon (;)

  – The notag keyword, which prevents the XML data handler from generating duplicate start tags (one for the business object and one for the attribute). The XML data handler creates an XML start and end tag for every business object attribute *unless* notag appears in the application-specific information for that attribute.

For example, suppose an XML element named Price has an attribute named Currency and requires data for Price:

```
<!ELEMENT Price (#PCDATA)>
<!ATTLIST Price Currency NMTOKEN #IMPLIED>
```

Because the Price element has an XML attribute, in its business object definition a business-object attribute must created for Currency. In addition, another attribute must exist to hold the Price data. The attribute for the Price data must specify notag in its application-specific information to prevent the data handler from creating a start and end tag for this attribute.

The Price child business object might look like this:

```
[BusinessObjectDefinition]
Name = Price
AppSpecificInfo = Price

   [Attribute]
   Name = Currency
   Type = String
   AppSpecificInfo = attr_name=Currency;type=attribute;
   ...
   [End]

   [Attribute]
   Name = Price
   Type = String
   AppSpecificInfo = Price;type=pcdata;notag
   ...
   [End]
```

In this case, the data handler does *not* generate a new XML element for the Price data but simply adds the data to the parent element.

For an XML element or attribute that contains special characters: Business object attributes representing XML elements or XML attributes with content that require escape processing must include the following tag in their application-specific information:

```
escape=true
```

An attribute requires escape processing if the attribute represents an XML element whose value contains any of the following special characters:

- single quotes (')
- double quotes (")
- ampersand (&)
- less-than sign (<)
- greater-than sign (>)

An attribute will *not* be escape-processed unless it contains the `escape=true` tag in its application-specific information. This tag must be placed at the end of any existing application-specific information. For example:

```
[Attribute]
Name=Data
Type=String
AppSpecificInfo=Price;type=pcdata;escape=true
[End]
```

If the attribute's application-specific information does *not* include the `escape` tag, the XML data handler checks the value of the `DefaultEscapeBehavior` property to determine whether to perform escape processing:

- If `DefaultEscapeBehavior` is `true`, the XML data handler performs escape processing on *all* attribute values.
- If `DefaultEscapeBehavior` is `false`, the XML data handler will *only* perform escape processing on attributes whose application-specific information contains the `escape` tag.

For an XML DOCTYPE declaration: If a business object attribute represents a document type declaration in the prolog, the application-specific information must include the following `type` tag:

```
type=doctype
```

For example, if a business object attribute named `DocType` represents a DOCTYPE element, its application-specific information is:

```
Name = DocType
AppSpecificInfo = type=doctype;
```

If the `DocType` attribute has the value:

```
DOCTYPE CUSTOMER "customer.dtd"
```

then the data handler generates the following XML:

```
<!DOCTYPE CUSTOMER "customer.dtd">
```

This application-specific information may also be used to include general entity declarations in the XML document. However, there is no explicit support for the inclusion of an internal DTD or parameter entity declarations. These can be included in the document by putting the entire text into the value of an attribute that has `type=doctype` in the application-specific information.

For a CDATA section: If a business object attribute represents a `CDATA` section within an XML document, the application-specific information must include the following `type` tag:

```
type=cdata
```

For example, if a business object attribute `UserArea` represents a `CDATA` section, its application-specific information is:

```
Name = UserArea
AppSpecificInfo = type=cdata;
```

For an XML comment: When the XML data handler converts a business object to an XML document, you can specify that it add comments to the XML document. To enable the data handler to add comments, take the following steps:

- Create in the business object definition the business object attribute (or attributes) that represents the comments.

  Note: The XML ODA does *not* automatically generate business object attributes for XML comments. You must manually add these attributes as described in "Manually creating business object definitions" on page 81.
- Include the following `type` tag in the attribute-level application-specific information:

  ```
  type=comment
  ```
- In the actual business object, specify the comment text as the value for this attribute.

For example, if a business object attribute named `Comment` represents an XML comment that the data handler should add to an XML document, the `Comment` attribute would appear as follows:

```
Name = Comment
AppSpecificInfo = type=comment;
```

If this attribute has the value `"Customer information update from application A"`, the following XML is generated:

```
<!--Customer information update from application A-->
```

For XML processing instructions: If a business object attribute represents a processing instruction, the application-specific information must include the following `type` tag:

```
type=pi
```

For example, if a business object attribute named `XMLDeclaration` represents the XML declaration in the prolog, the application-specific information is:

```
Name = XMLDeclaration
AppSpecificInfo = type=pi;
```

If the attribute has the value:

```
xml version = "1.0"
```

then the XML data handler generates the following XML:

```
<?xml version="1.0"?>
```

Creating business object definitions from DTDs

A DTD describes the format of an XML document. Therefore, the DTD is very useful for obtaining information needed for the business object definition. To translate the structure information in the DTD to a business object definition, you can use the XML Object Discovery Agent (ODA). For information on XML ODA, see "Using an XML ODA to create business object definitions" on page 80.

Note: Previous versions of the XML data handler included two tools to create business object definitions from DTDs: Edifecs SpecBuilder and the deprecated XMLBorgen utility. The XML ODA replaces both these tools have been replaced in functionality.

Supported DTD structures

The XML ODA supports structures of a DTD including:

- Entities — User-defined entities of the following form are recognized and substituted wherever referenced:

  ```
  <!ENTITY % name value>
  ```

  Note: The XML ODA removes newline characters (\n), carriage returns (\r), or tab characters (\t) if the characters occur between the ENTITY tag and its content, or between an element or attribute tag and the content of the tag.
- External DTDs — The XML ODA supports references to external DTDs (where one DTD refers to one or more other DTDs). They can resolve external DTDs *only* if they are on the local file system; they cannot access a URL to find them. Both these tools always try to find references to the external DTDs on the local file system; they do *not* ignore them.
- ANY directive — The XML ODA creates a String attribute for an element with ANY as its content. For example, if the DTD has the following construct:

  ```
  <!ELEMENT SCENE (ANY) >
  ```

  The corresponding business object definition is:

  ```
  [Attribute]
  Name = SCENE
  Type = String
  Cardinality = 1
  MaxLength = 255
  IsKey = false
  IsForeignKey = false
  IsRequired = true
  AppSpecificInfo = SCENE;type=pcdata;
  [End]
  ```
- Prolog — The XML data handler populates prolog information such as the DOCTYPE and XML declaration, as long as the business object definition contains attributes that correspond to those elements. However, the data handler populates *only* the name of the DOCTYPE; it does *not* populate other possible meta information.

Unsupported DTD structures

The XML ODA is capable of processing most DTDs. However, it does *not* support the following DTD structures:

- Conditional sections — These sections have the following structure:

  ```
  <![INCLUDE[ information to be included ]]>
  <![IGNORE[ information to be ignored ]]>
  ```
- Namespaces — A tag of the form xxx:yyy is treated simply as a tag xxx:yyy, and not as a tag yyy belonging to a namespace xxx.

XML documents that use schema documents

An *XML Schema* is a data model for XML documents that uses a *schema document* (.xsd extension) to define the template (schema) of an XML document. Unlike a DTD, a schema document uses the same syntax as an XML document to describe the schema. The business object definitions that represent the schema of an XML document use information in the schema document to preserve and record the

document's structure. This section provides the following information about deriving structure information for a business object definition from a schema document:

- "Requirements for business object definitions based on schema documents"
- "Creating business object definitions from schema documents" on page 78

Requirements for business object definitions based on schema documents

To ensure that business object definitions that represent schema documents conform to the requirements of the XML data handler, use the guidelines in this section, which involve:

- "Business object structure for schema documents"
- "Business object attribute properties for schema documents" on page 63
- "Application-specific information for XML components in schema documents" on page 64

Business object structure for schema documents

A business object that is processed by the XML data handler using business object definitions based on schema documents must follow these rules:

- The schema document must have the following required business object definitions:
  - A top-level business object definition represents the `schema` element.
  - A root-element business object definition represents the schema document's root element. A regular, mixed, or wrapper business object definition represents the root element in the schema document.

  For more information, see "Required business object definitions for schema documents" on page 56.
- A regular, mixed, or wrapper business object definition can represent contained XML components as well.

An example schema document for an XML document is shown in Figure 18. The schema document is named `Order`, and it contains elements that correspond to an application `Order` entity. This sample schema document describes the same business object structure as the one that the sample DTD document described. Figure 13 on page 44 shows the structure of a business object definition that might be created to correspond to an XML document associated with the `Order` schema document.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified">
        <xs:complexType name="AccessoryType">
                <xs:sequence>
                        <xs:element ref="Quantity"/>
                        <xs:element ref="Type"/>
                </xs:sequence>
                <xs:attribute name="Name" type="xs:string" use="required"/>
        </xs:complexType>
        <xs:element name="Order">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="Unit" type="UnitType"
                    maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="PartNumber" type="xs:string"/>
        <xs:element name="Price" type="xs:string"/>
        <xs:element name="Quantity" type="xs:string"/>
        <xs:element name="Type" type="xs:string"/>
        <xs:complexType name="UnitType">
                <xs:sequence>
                        <xs:element ref="PartNumber" minOccurs="0"/>
            <xs:element ref="Quantity"/>
            <xs:element ref="Price"/>
            <xs:element name="Accessory" type="AccessoryType" minOccurs="0"
                   maxOccurs="unbounded"/>
                </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Figure 18. Sample XML schema document

Required business object definitions for schema documents: To represent a schema document requires at least the two business object definitions described in "Business object structure" on page 35. For a schema document, these business object definitions have the following additional requirements:

- The *top-level* business object represents the `schema` element and must contain the following:
  - An attribute named `XMLDeclaration` to represent the XML version

    This attribute must has the `type=pi` tag in its application-specific information. For more information, see "For XML processing instructions" on page 76.
  - An attribute to represent the root element in the schema document

    As described in "Business object structure" on page 35, this attribute must have as its type a single-cardinality business object, whose type is the business object definition for the root element. The application-specific information must list the name of this element with the `elem_name` tag. For more information, see "For XML elements" on page 72.
- A *root-element* business object definition represents the schema document's root element. Because a schema document can have several XML components defined at the global level, you can tell the XML ODA which element to consider the root element through its `Root` configuration property. The root-element business object definition can contain the following attributes:
  - A `schemaLocation` attribute (optional) to represent where the schema document (or documents) resides

Whether the XML ODA generates a schemaLocation attribute in the top-level business object definition depends on the setting of its DocTypeOrSchemaLocation configuration property. For more information, see "For XML schema locations" on page 76.

– An attribute for each of the XML components in the root element

Note: For a list of general business object requirements, see "Requirements for business object definitions" on page 34.

Both these required business object definitions require business-object-level application-specific information that defines the target namespace and any component-name qualifications. For more information, see "Business-object-level application-specific information" on page 64.

The XML schema element in Figure 18 on page 56 is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified">
```

Figure 19 shows the top-level business object definition, TopLevel, which represents this schema element.

```
[BusinessObjectDefinition]
Name=TopLevel
AppSpecificInfo=elem_fd=qualified;attr_fd=unqualified
...
   [Attribute]
   Name=XMLDeclaration
   Type=String
   AppSpecificInfo=type=pi;
   ...
   [End]
   [Attribute]
   Name=Order
   Type=TopLevel_Order
   AppSpecificInfo=elem_name=Order;
   ...
   [End]
...
[End]
```

Figure 19. Sample top-level business object definition

If the XML ODA generated the top-level business object definition in Figure 19, the following ODA configuration properties would have been set:

ODA configuration property	Value of property
BOPrefix	Not set
TopLevel	"TopLevel"
Root	"Order"
DoctypeorSchemaLocation	true

For an example of a root-element business object definition, see Figure 26 on page 69.

Regular business object definitions based on schema documents: A *regular* business object definition represents any of the following XML constructs:

- An XML element that contains a complex type (either named or anonymous) that contains a `sequence` group of child elements.

  Each child element in the `sequence` group is represented as an attribute in the business object definition. For more information, see "For XML elements within a complex type" on page 73.

  Note: If the complex type contains a `choice` or an `all` group, it must be represented by a wrapper business object definition. For more information, see "Wrapper business object definitions based on schema documents" on page 60.
- An XML element that contains attributes

In these types of business object definition, the business-object-level application-specific information does *not* require any special information. The attributes of a regular business object definition represent the elements defined within the XML complex type.

Note: For complex types that contain a `sequence` group of child elements, each child element in the sequence group is represented as an attribute in the business object definition.

For example, suppose an XML element named Unit is defined as:

```
<xsd:element name="Unit">
    <xsd:complexType>
    ...
    </xsd:complexType>
</element>
```

The following business object definition represents the Unit element:

```
[BusinessObjectDefinition]
Name = MyApp_Unit
AppSpecificInfo =
   [Attribute]
   ...
```

Mixed business object definitions based on schema documents: A *mixed* business object definition represents a mixed XML element, one that contains mixed content of character data and other subelements. A schema document describes a mixed-type XML element as a complex type with the `mixed` attribute set to `true`, as follows:

```
<xsd:complexType mixed="true">
    <xsd:sequence>
        <xsd:element name="subElement1" type="subElementType"/>
        ...
    </xsd:sequence>
</xsd:complexType>
```

Note: Mixed business objects, as described in this section, are used for a repeating list of elements *and* character data. If your choice list does *not* contain any character data, use a wrapper business object as described in "Qualified component names" on page 69.

Because this complex type sets the `mixed` attribute to `true`, it can contain character data in addition to the one or more subelements it defines. If the `mixed` attribute is set to `false`, no character data is permitted in the complex type.

For example, Figure 20 shows a mixed-type XML element defined in a schema document.

```
<xsd:complexType name="Cust" mixed="true">
    <xsd:sequence>
        <xsd:element name="Name"/>
        <xsd:element name="Address"/>
        <xsd:element name="Phone"/>
    </xsd:sequence>
</xsd:complexType>
```

Figure 20. Sample schema document for a mixed-type XML element

To represent a mixed-type XML element requires the following two business object definitions:

- The parent business object definition

  This parent business object definition contains a single attribute that represents a multiple-cardinality business object array. This attribute has as its type the business object definition for the associated wrapper business object. The parent business object definition includes the following application-specific information:

  – The business-object-level application-specific information consists of the following components:

    - The name of the associated mixed-type XML element followed by a semicolon (;)
    - The tag `type=MIXED`

  – The multiple-cardinality attribute has the following tag in its application-specific information:

    (*mixedTypeElement*|*subElement1*|...|*subElementN*)

    where:

    - *mixedTypeElement* is the name of the associated mixed-type XML element
    - *subelement1*...*subElementN* correspond to each of the subelements defined in the complex type

    The pipe (|) character must separate each of the subelements and the entire tag must be enclosed in parentheses.

- A wrapper business object definition

  The wrapper business object definition contains attributes for the mixed data:

  – One attribute for the character data, which requires the `type=pcdata` tag (to indicate character data) as well as the `notag` tag (to indicate that the data is associated with the current element, not a separate element).

  – One attribute for each of the subelements defined in the mixed-type element. Each attribute requires the `type=pcdata` tag (to indicate that it represents a simple type).

  For more information on the `type=pcdata` tag, see "For XML elements within a complex type" on page 73.

  Note: This wrapper business object definition does *not* require any application-specific information at the business-object level.

If the business object definition, `MyApp_Cust`, represents the `Cust` mixed-type element in Figure 20, its application-specific information is as follows:

```
[BusinessObjectDefinition]
Name = MyApp_Cust
AppSpecificInfo = type=MIXED;

   [Attribute]
   Name=Cust_wrapper1
   Type=MyApp_CustWrapper1
   Cardinality=n
   AppSpecificInfo=(Cust|Address|Phone)
   ...
   [End]
```

Wrapper business object definitions based on schema documents: A *wrapper* business object definition represents a repeating choice list. This type of business object definition is needed when an XML element has children that can appear in any order and be of any cardinality. A wrapper business object preserves the order and cardinality of the child elements in a particular XML document.

Note: Wrapper business objects, as described in this section, are used for a repeating choice list that contains elements, *not* for character data. If your choice list contains any character data, use a mixed business object. For more information, see "Mixed business object definitions based on schema documents" on page 58.

A schema document can describe a choice-list XML element as a complex type that contains either of the following model groups:

- A `choice` group, which defines an unordered list of elements in which only one element must appear:

  ```
  <xsd:complexType>
      <xsd:choice>
      ...
      </choice>
  </complexType>
  ```

  To represent an XML element with a complex type that contains a `choice` group, the XML data handler expects the same hierarchical business object definition that it does for a choice-list XML element defined in a DTD:

  – A parent business object definition that contains a single attribute with *multiple cardinality* whose type is the wrapper business object

  – A wrapper business object definition that contains an attribute for each subelement in the `choice` group.

  These business object definitions contain business-object-level application-specific information in the same format as the choice-list XML element for a DTD. For more information, see "Wrapper business object definitions based on DTDs" on page 46.

- An `all` group, which defines an unordered list of elements in which elements can appear no more than once each:

  ```
  <xsd:complexType>
      <xsd:all>
      ...
      </xsd:all>
  </xsd:complexType>
  ```

  To represent an XML element with a complex type that contains an `all` group, the XML data handler expects a hierarchical business object definition with the following business object definitions:

  – A parent business object definition that contains a single attribute with *multiple cardinality* whose type is the wrapper business object

– A wrapper business object definition that contains an attribute for each subelement in the `all` group

  These business object definitions contain business-object-level application-specific information in the same format as the choice-list XML element for a DTD. For more information, see "Wrapper business object definitions based on DTDs" on page 46.

To indicate cardinality, these model groups support occurrence constraints with the `minOccurs` and `maxOccurs` attributes. For more information, see Table 20 on page 63.

As an example, the XML element definition in a schema document might be:

```
<xsd:element name="CUST">
    <xsd:complexType>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="U"/>
            <xsd:element ref="I"/>
            <xsd:element ref="B"/>
        </xsd:choice>
    </xsd:complexType>
</xsd:element>
```

This element contains three subelements that are optional (although one subelement out of the list *must* occur) and that can appear in any order. Figure 14 on page 46 shows an XML document of this type. Figure 16 on page 47 shows the parent business object definition for this XML document. Figure 17 on page 48 shows the wrapper business object definition.

Type substitution in business object definitions based on schema documents: *Type substitution* enables derived types to appear in place of their base types in individual XML document instances. When type substitution occurs, an element conforming to a declaration with one data type can have any data type that either extends or restricts it. In the following schema definition, ShirtType and HatType are derived types of the basic ProductType:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="items" type="ItemsType"/>
<xsd:complexType name="ItemsType">
  <xsd:sequence>
            <xsd:element ref="product" maxOccurs="unbounded"/>
          </xsd:sequence>
</xsd:complexType>
<xsd:element name="product" type="ProductType"/>
<xsd:complexType name="ProductType">
    <xsd:sequence>
          <xsd:element name="number" type="xsd:string"/>
            <xsd:element name="name" type="xsd:string"/>
      </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ShirtType">
    <xsd:complexContent>
    <xsd:extension base="ProductType">
          <xsd:sequence>
                <xsd:element name="size" type="xsd:string"/>
                <xsd:element name="color" type="xsd:string"/>
          </xsd:sequence>
    </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="HatType">
    <xsd:complexContent>
    <xsd:extension base="ProductType">
      <xsd:sequence>
                <xsd:element name="size" type="xsd:string"/>
              </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:schema>
```

Figure 21. Sample schema with type substitution

Based on the schema above, the following XML document is valid:

```
<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<product>
      <number>999</number>
      <name>Special Seasonal</name>
</product>
<product xsi:type="ShirtType">
      <number>557</number>
      <name>Short-Sleeved Linen Blouse</name>
      <size>M</size>
      <color>blue</color>
</product>
<product xsi:type="HatType">
      <number>443</number>
      <name>Four-Gallon Hat</name>
      <size>L</size>
</product>
</items>
```

Figure 22. Derived types in an XML document

Wherever ProductType occurs, its derived types, ShirtType and HatType, indicated by the xsi:type attribute, can appear instead. To represent XML documents in which type substitution occurs, the XML data handler creates a wrapper business object as a child attribute of the XML document. This wrapper business object has

child attributes corresponding to the compex type (ProductType in Figure 21 on page 62) and its derived types (ShirtType and HatType). Table 18 and Table 19 show the business object definitions that would be generated from the XML schema above.

Table 18. Business object definitions for ItemsType

Attribute Name	Type	Cardinality	ASI
Product	ProductTypeWrapper	N	(product);typeSub=true

Table 19. Business object definitions for ProductTypeWrapper

Attribute	Type	Cardinality	ASI
ProductType	ProductType	1	Elem_name=product; xsiType=ProductType
ShirtType	ShirtType	1	Elem_name=product; xsiType=ShirtType;
HatType	HatType	1	Elem_name=product; xsiType=HatType;

Business object attribute properties for schema documents

When the business object definitions for an XML document are based on schema documents, the business object attribute properties have the restrictions discussed in "Business object attribute properties" on page 36. In addition, the schema-document syntax can determine the ″required-ness″ of a business object attribute. The ″required-ness″ is a combination of factors, including cardinality and whether the attribute is a key, that determines whether the XML data handler requires the attribute. If an attribute is required, its Required attribute property must be set to true.

The setting of the Required attribute property depends on the XML element and attribute specifications, as well as the settings of the Cardinality, Key, and Foreign Key attribute properties, as follows:

- The cardinality of a business object attribute is determined by the Occurrence indicator in the schema document. This cardinality affects whether the attribute is required. Table 20 outlines the cardinality and ″required-ness″ for possible combinations of element declarations in a schema document.

Table 20. Cardinality and "Required-ness" for a schema document

Schema fragment occurrence indicator	Cardinality	Required
None specified	1	Yes
maxOccurs > 1	N	Yes
maxOccurs = ″unbounded″	N	Yes
minOccurs=0	No effect	No
minOccurs>1	N	Yes

- Whether a business object attribute is required is also determined by the `use` attribute in the schema document. Table 21 outlines the ″required-ness″ for possible values of the `use` attribute.

Table 21. "Required-ness" for a schema document

Schema fragment occurrence attribute: use	Cardinality	Required
None specified	No effect	No
`use=required`	No effect	Yes

- Whether a business object attribute is a primary or foreign key is determined by the `id` attribute in the schema document. The presence of a key affects whether the attribute is required. Table 22 outlines how value of the `id` attribute affects the business object attribute's ″required-ness″.

Table 22. Keys and ″Required-ness″ for a schema document

Schema fragment attribute: id	Key	Required	Comment
`id=ID`	Yes	No	

Application-specific information for XML components in schema documents

This section provides the following information on the application-specific information format for business object definitions based on schema documents:

- "Business-object-level application-specific information"
- "Attribute application-specific information" on page 71

Business-object-level application-specific information: The XML data handler uses the following types of business object definitions to represent the different kinds of root XML elements defined in a schema document. These types of business object definitions are distinguished by the application-specific information at the business object level.

Table 23. Tags for business-object-level application-specific information

Tag in application-specific information	Description	For more information
`target_ns`	Specifies the target namespace of the schema document	"Schema namespaces"
`attr_fd`	Specifies whether attribute names are qualified or unqualified.	"Qualified component names" on page 69
`elem_fd`	Specifies whether the names of locally declared elements are qualified or unqualified.	"Qualified component names" on page 69

Note: Business-level application-specific information can also include the `type=MIXED` tag. For more information, see "Mixed business object definitions based on schema documents" on page 58.

Schema namespaces: Unlike DTDs, schema documents require definition of at least one namespace. A *namespace* provides a context for the names of elements, element types, and attributes within an XML document. A namespace is a Uniform Resource Identifier (URI), which includes HTTP, FTP, as well as other kinds of paths. Table 24 shows the namespaces that a schema document can declare to be able to resolve the references between schema components.

Table 24. XML namespaces

Namespace	Description	Name	Common prefix
XML Schema Namespace	Defines every component used in the XML Schema Definition Language (XSDL), such as `element`, `schema`, and `simpleType`.	`http://www.w3.org/2001/XMLSchema`	`xsd`, `xs`
XML Schema Instance Namespace	Defines four attributes associated with the schema instance: `type`, `nil`, `schemaLocation`, `noNamespaceSchemaLocation`	`http://www.w3.org/2001/XMLSchema-instance`	`xsi`
User-defined namespace	Defines every component declared or defined by a global declaration (such as `element`, `attribute`, `type`, or `group`). **Note:** Locally declared elements may or may not use the target namespace. For more information, see "Qualified component names" on page 69.	User-defined	User-defined

Note: The XML ODA supports schema documents with multiple target namespaces.

Every schema document can declare one *target namespace*, which identifies the namespace to which global components (elements, attributes, types, or groups) belong, with the `targetNamespace` tag. If the schema element includes the `targetNamespace` tag, each business object definition generated for that schema document must contain the `target_ns` tag in its application-specific information to specify the target namespace declared for the XML document:

`target_ns=`*`URI address for target namespace`*

The `target_ns` tag must exist in the business-object-level application-specific information for *all* business object definitions, as follows:

- For the top-level business object definition, the value of the `target_ns` tag specifies the value that the `targetNamespace` attribute of the XML `schema` element specifies.
- Each business object definition that represents an XML element must also include the `target_ns` tag in its application-specific information. Because every global component (such as element, attribute, or type) belongs to its schema document's target namespace, the business object definitions that represent these components also specify the schema document's target namespace.

Note: Previous versions of the XML data handler expected the top-level business object definition to have attributes for the namespace prefixes and to use the appropriate `type=defaultNS` and `type=xmlns` tags in the attribute-level application-specific information. This mechanism for defining namespace prefixes has been replaced, although the XML data handler continues to support it for backward compatibility with existing business object definitions. New business object definitions should use the `target_ns` tag as described in this section. The XML ODA has been modified to use the `target_ns` tag.

For example, the schema document in Figure 23 defines the XML Schema Namespace (with the xsd prefix) and a target namespace (which is the default namespace).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.ibm.com/ns1" xmlns="http://www.ibm.com/ns1">
   <xsd:complexType name="TaxInfoType">
      <xsd:sequence>
         <xsd:element name="SSN" type="string">
         </xsd:element>
         <xsd:element name="State" type="string">
         </xsd:element>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:element name="Customer">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="TaxInfo" type="TaxInfoType">
            </xsd:element>
            <xsd:element name="BillTo" type="xsd:string">
            </xsd:element>
         </xsd:sequence>
         <xsd:attribute name="Name" type="xsd:string">
         </xsd:attribute>
         <xsd:attribute name="ID" type="xsd:string">
         </xsd:attribute>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Figure 23. The Schema1.xsd sample schema document

The XML ODA generates three business object definitions for this schema document: BOPrefix_TopLevel, BOPrefix_TopLevel_Customer, and BOPrefix_TopLevel_TaxInfoType (where BOPrefix and TopLevel are the values of these ODA configuration properties). All three of these business object definitions have the following in their business-object-level application-specific information:

```
target_ns=http://www.ibm.com/ns1;elem_fd=unqualified;attr_fd=unqualified
```

Note: Because of the schema element in Figure 23 includes neither the elementFormDefault nor the attributeFormDefault attribute, this application-specific information includes the elem_fd and attr_fd tags set to unqualified. For more information, see "Qualified component names" on page 69.

One schema document can only define one target namespace. However, it can include elements and attributes defined in another schema document's target namespace by using the import element.

Figure 24 shows a schema document that is based on the Schema1.xsd document defined in Figure 23. This schema document imports the ns2 namespace, which declares the TaxInfoType complex type, the BillTo element, and the Name attribute.

```
<?xml version "1.0" encoding="UTF-8"?>
<xsd:schema smlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.example.com/ns1" xmlns="http://www.example.com/ns1"
   attributeFormDefault="qualified" elementFormDefault="qualified"
   xmlns:ns2="http://www.example.com/ns2">
   <xsd:import schemaLocation="Schema2.xsd"
      namespace="http://www.example.com/ns2"/>
   <xsd:element name="Customer2">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="TaxInfo" type="ns2:TaxInfoType">
            </xsd:element>
            <xsd:element ref="ns2:BillTo">
            </xsd:element>
         </xsd:sequence>
         <xsd:attribute ref="ns2:Name">
         </xsd:attribute>
         <xsd:attribute name="ID" type="xsd:string">
         </xsd:attribute>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Figure 24. Importing a target namespace

The business object definition for the root element, Customer2, must specify this alternate namespace for its attributes that represent the TaxInfo, BillTo, and Name XML components, as follows:

- The TaxInfo attribute has as its type a business object definition that represents the TaxInfoType, which is defined in the ns2 (http://www.example.com/ns2) namespace (see Figure 25 on page 68).
- The BillTo attribute has the elem_ns tag in its application-specific information to specify the ns2 namespace as the source of the associated BillTo element.
- The Name attribute has the attr_ns tag in its application-specific information to specify the ns2 namespace as the source of its associated XML attribute, Name.

Figure 25 shows the schema document that defines the TaxInfoType, BillTo, and Name XML components in the ns2 namespace.

```
<?xml version "1.0" encoding="UTF-8"?>
<schema smlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.example.com/ns2"
   attributeFormDefault="qualified" elementFormDefault="qualified"
   xmlns:ns2="http://www.example.com/ns2">
   <complexType name="TaxInfoType">
      <sequence>
         <element name="SSN" type="string">
         </element>
         <element name="State" type="string">
         </element>
      </sequence>
   </complexType>
   <attribute name="Name" type="string"
   </attribute>
   <complexType name="AddressType">
      <sequence>
         <element name="Zip" type="string">
         </element>
         <element name="Street" type="string">
         </element>
      </sequence>
   </complexType>
   <element name="BillTo" type="ns2:AddressType">
   </element>
</schema>
```

Figure 25. Defining the second namespace

Figure 25 shows the business object definition for the `Customer2` root element.

```
[BusinessObjectDefinition]
Name=TopLevel_Customer2
AppSpecificInfo=target_ns=http://www.example.com/ns1;elem_fd=qualified;
   attr_fd=qualified
...
   [Attribute]
   Name=Name
   Type=String
   AppSpecificInfo=attr_name=Name;type=attribute;attr_ns=http://www.example.com/ns2
   ...
   [End]
   [Attribute]
   Name=ID
   Type=String
   AppSpecificInfo=attr_name=ID;type=attribute
   ...
   [End]
   [Attribute]
   Name=schemaLocation
   Type=String
   AppSpecificInfo=attr_name=schemaLocation;type=xsischemalocation
   ...
   [End]
   [Attribute]
   Name=TaxInfo
   Type=TopLevel_TaxInfoType
   AppSpecificInfo=elem_name=TaxInfo
   ...
   [End]
   [Attribute]
   Name=BillTo
   Type=TopLevel_AddressType
   AppSpecificInfo=elem_name=BillTo;elem_ns=http://www.example.com/ns2
   ...
   [End]
...
[End]
```

Figure 26. Sample root-element business object definition

Qualified component names: Within the XML document, names of components associated with a namespace are either qualified or unqualified, as follows:

- Unqualified names do *not* contain a prefix and are *not* part of any namespace.
- Qualified names are one of the following:
  - The component name contains a prefix associated with a namespace.

    You can assign a prefix to one or more namespaces. You declare a namespace's prefix with the `xmlns:`*`prefix`* tag, where *prefix* is the declared prefix. In the component definition within the schema document, these component names are *qualified* because they have a prefix prepended to the component name (*prefix*:*componentName*).
  - The names do not contain a prefix but are part of the default namespace (elements only).

    The *default namespace* specifies the namespace to associate with components that do *not* include a prefix in their component names. You declare the default namespace with the `xmlns` tag.

    For the XML data handler to correctly handle XML-to-business-object conversion, the namespaces for the XML document and the schema document must match, as follows:

- If a schema document specifies a default namespace, the XML document must also specify default namespace.
- If a schema document does *not* have a default namespace, the XML document cannot have a default namespace.

The `elementFormDefault` attribute of the `schema` element specifies whether the names of locally declared elements are qualified. By default, locally declared elements are unqualified and belong to the default namespace. The value of the `elementFormDefault` attribute determines the value of the `elem_fd` tag in the business-object-level application-specific information, as Table 25 shows.

Table 25. Setting the elem_fd tag

Value of elementFormDefault	Value of elem_fd tag
`"unqualified"`(or attribute is not specified at all)	`elem_fd=unqualified`
`"qualified"`	`elem_fd=qualified`

For example, the schema document in Figure 23 on page 66 does *not* contain the `elementFormDefault` attribute in its `schema` element. Therefore, the business-object-level application-specific information in *all* the business object definitions for this schema document (`BOPrefix_TopLevel`, `BOPrefix_TopLevel_Customer`, and `BOPrefix_TopLevel_TaxInfoType`, where `BOPrefix` and `TopLevel` are the values of these ODA configuration properties) contains the tag:

```
elem_fd=unqualified
```

Note: The business-object-level application-specific information for these three business object definitions would contain this same tag if the schema document's `schema` element included the attribute:

```
elementFormDefault="unqualified"
```

If an individual XML element includes the `form` attribute, this value of the `form` attribute overrides any setting of the `elementFormDefault` attribute.

The `attributeFormDefault` attribute of the `schema` element specifies whether the names of elements are qualified. By default, attribute names are unqualified and do not belong to any namespace. The value of the `attributeFormDefault` attribute determines the value of the `attr_fd` tag in the business-object level application-specific information in the same way as the value of the `elementFormDefault` attribute determines the value of the `elem_fd` tag, as Table 26 shows.

Table 26. Setting the attr_fd tag

Value of attributeFormDefault	Value of elem_fd tag
`"unqualified"`(or attribute is not specified at all)	`attr_fd=unqualified`
`"qualified"`	`attr_fd=qualified`

For example, the schema document in Figure 23 on page 66 does *not* contain the `attributeFormDefault` attribute in its `schema` element. Therefore, the business-object-level application-specific information in *all* the business object definitions for this schema document (`BOPrefix_TopLevel`,

BOPrefix_TopLevel_Customer, and BOPrefix_TopLevel_TaxInfoType, where BOPrefix and TopLevel are the values of these ODA configuration properties) contains the tag:

```
attr_fd=unqualified
```

Note: The business-object-level application-specific information for these three business object definitions would contain this same tag if the schema document's schema element included the attribute:

```
attributeFormDefault="unqualified"
```

Attribute application-specific information: The attribute of a business object definition can represent the following XML components:

- "For XML elements" on page 72
- "For XML elements within a complex type" on page 73
- "For an XML attribute" on page 75
- "For XML processing instructions" on page 76
- "For an XML comment" on page 76
- "For an XML element or attribute that contains special characters" on page 76
- "For XML schema locations" on page 76

Table 27 shows the tags for attribute-level application-specific information for these different XML components along with the sections in this manual that describe these tags in more detail.

Table 27. Tags for attribute application-specific information

Representation of business object attribute	Application-specific information	For more information
An XML element	elem_name=*name of XML element* elem_ns=*namespace for element's definition* elem_fd=*value of* form *attribute*	"For XML elements" on page 72
An XML element within a complex type	type=pcdata	"For XML elements within a complex type" on page 73
An attribute for an XML element	attr_name=*name of XML attribute* type=attribute attr_ns=*namespace for attribute's definition* attr_fd=*value of* form *attribute*	"For an XML attribute" on page 75
An XML element or attribute whose content includes special characters	escape=true	"For an XML element or attribute that contains special characters" on page 76
A comment to be added to the XML document	type=comment	"For an XML comment" on page 76
A processing instruction	type=pi	"For XML processing instructions" on page 76
The schemaLocation or noNamespaceSchemaLocationattribute for an XML instance.	type=xsischemalocation type=xsinoNSlocation	"For XML schema locations" on page 76

Note: Attribute application-specific information can also include a tag of the form `( a | b | c )` to specify a multiple-cardinality attribute that represents a repeating choice. For more information, see "Wrapper business object definitions based on schema documents" on page 60.

For XML elements: If a business object attribute represents an XML element, its application-specific information must include the `elem_name` tag to identify the associated element:

```
elem_name=name of XML element
```

XML element names can contain special characters (such as periods and hyphens). However, the names of business object attributes cannot contain these special characters. Therefore, the name of the XML element must be specified in the `elem_name` tag. To name the business object attribute, the XML ODA removes any special characters in the XML element's name.

A business object attribute can represent an XML element in the following cases:

- If the XML element is (or contains) an XML complex type

  In this case, the business object attribute is a *complex* attribute whose data type is the business object definition that represents the XML complex type. Its `elem_name` tag (in the attribute's application-specific information) contains the name of the XML element (or complex type).
- If the XML element is part of an XML complex type

  In this case, the business object attribute is a *simple* attribute of type `String`. Its application-specific information includes the `elem_tag` (which contains the name of the XML element within the complex type) and the `type=pcdata` tag. For more information, see "For XML elements within a complex type" on page 73.

A business object attribute that represents an XML element can also include the following tags in its application-specific information:

- The `elem_fd` tag specifies the setting of the XML element's `form` attribute, which indicates whether the names of locally declared element are qualified or unqualified. If an XML element has specified the attribute `form="qualified"`, the value of elem_fd is set to the value of the form attribute.

  For example, suppose a locally declared XML element has the following definition:

  ```
  <xsd:element ref="Name" form="qualified"></xsd:element>
  ```

  Its associated business object attribute would have the following format:

  ```
  [Attribute]
  Name=ns2:Name
  Type=String
  AppSpecificInfo=elem_name=Name;elem_fd=qualified;
  ...
  ```

  If an XML element does *not* specify the `form` attribute, the value of the `elementFormDefault` attribute (on the `schema` element) determines whether element names are qualified. For more information, see "Qualified component names" on page 69.
- The `elem_ns` tag specifies the target namespace for the XML element, if this namespace is *different* from the schema document's target namespace. This tag is required when a schema document uses multiple namespaces. If the XML element referenced in one schema document is defined in the target namespace of *some other schema document*, this tag lists the name of that namespace.

  For example, suppose an XML element within a complex type has the following definition:

```
<xsd:element ref="ns2:BillTo"></xsd:element>
```

Its associated business object attribute would have the following format:

```
[Attribute]
Name=BillTo
Type=String
AppSpecificInfo=elem_name=BillTo;elem_ns=http:/www.imb.com/ns2;
...
```

For a complete example of a schema document that includes XML elements defined in a second namespace, see "Schema namespaces" on page 64.

For XML elements within a complex type: If a business object definition represents an XML complex type (`complexType`), the contents of this complex type are represented by simple (`String`) attributes within this business object definition. These business object attributes' application-specific information *must* include the `elem_name` tag to identify the name of the element.

Note: For more information on this tag, see "For XML elements" on page 72.

Within an XML complex type, a business object attribute can represent the kinds of complex-type content shown in Table 28.

Table 28. Contents of an XML complex type

Type of complex-type element	Description	Attribute application-specific information
Simple XML element	An element that contains *only* character content. It cannot contain any other elements or XML attributes. It can occur only within a complex type	`type=pcdata`
Simple content	Only character data (no element)	`type=pcdata;notag`
XML element with attributes	An element that contains *both* character data *and* some combination of subelements and attributes	None. This type of XML element must be represented with a business object definition, not as a single business object attribute. For more information, see "Regular business object definitions based on schema documents" on page 58.

As an example of the use of the `type=pcdata` tag, the schema document in Figure 23 on page 66 contains a definition for the `TaxInfoType` complex type, which contains only simple XML elements. The business object definition for the TaxInfoType XML complex type (`BOPrefix_TopLevel_TaxInfoType`, where `BOPrefix` and `TopLevel` are the values of the respectively named ODA configuration properties) would contain two attributes. Each attribute would have the name of the associated XML element in its application-specific information, as Figure 27 shows. Because this complex type contains only simple XML elements (no subelements or attributes), the corresponding business object attributes must also contain the `type=pcdata` tag in their application-specific information.

```
[BusinessObjectDefinition]
Name=BOPrefix_TopLevel_TaxInfoType
AppSpecificInfo=target_ns=;elem_fd=unqualified;attr_fd=unqualified
...
[End]

[Attribute]
Name=SSN
Type=String
AppSpecificInfo=elem_name=SSN;type=pcdata
...
[End]

[Attribute]
Name=State
Type=String
AppSpecificInfo=elem_name=SSN;type=pcdata
...
[End]
```

Figure 27. Sample business object definition for XML complex type with simple elements

As an example of the notag keyword used in conjunction with the type=pcdata tag, suppose an XML element named Price is the PriceType complex type, which has only simple content; that is, it contains only character data. In this case, the simpleContent element defined an attribute named Currency and requires data for Price:

```
<xsd:element name="Price" type="PriceType">
    <xsd:complexType name="PriceType">
        <xsd:simpleContent>
            <xsd:extension base="xsd:decimal">
                <xsd:attribute name="Currency" type="xsd:NMTOKEN"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</element>
```

The business object definition for the PriceType complex type contains an attribute to hold the character data associated with simple content. The application-specific information for this attribute must contain the following:

```
type=pcdata;notag
```

The notag keyword prevents the XML data handler from generating duplicate start tags (one for the business object definition and one for the attribute). The XML data handler creates an XML start and end tag for every business object attribute *unless* notag appears in the application-specific information for that attribute.

The Price child business object definition might look like this:

```
[BusinessObjectDefinition]
Name = Price
AppSpecificInfo =

[Attribute]
Name = Currency
Type = String
AppSpecificInfo = attr_name=Currency;type=attribute;
...
[End]

[Attribute]
Name = Price
```

```
Type = String
AppSpecificInfo = elem_name=Price;type=pcdata;notag
...
[End]
```

A business object attribute must exist to hold the `Price` data. The attribute for the `Price` data must specify `notag` in its application-specific information to prevent the data handler from creating a start and end tag for this attribute. In this case, the data handler does *not* generate a new XML element for the `Price` data but simply adds the data to the parent element.

In addition, the business object definition must contain another attribute must exist to hold the value of the `Currency` attribute. If the simple content contains attributes, the business object definition must also contain an attribute for each XML attribute. The attribute's application-specific information must include the `type=attribute` tag. For more information, see "For an XML attribute."

For an XML attribute: When a business object definition represents an XML element or complex type, any attributes that the schema document declares for this element are represented as attributes within the business object definition. When a business object attribute represents an attribute of an XML element, its application-specific information *must* include the following tags:

- The `attr_name` tag:

  `attr_name=`*`name of XML attribute`*
- The `type` tag:

  `type=attribute`

Note: For more information on these tags, see "For an XML attribute" on page 50. For an example of the use of the `type=attribute` tag, see "For XML elements within a complex type" on page 73.

A business object attribute that represents an XML attribute can also include the following tags in its application-specific information:

- The `attr_fd` tag specifies the setting of the XML attribute's `form` attribute, which indicates whether the attribute names are qualified or unqualified. If an XML attribute has the attribute `form="qualified"` for `form="unqualified"` specified, the attr_fd has a value specified in the ″form″ attribute.

  For example, suppose an XML attribute has the following definition:

  ```
  <xsd:attribute ref="Name" form="qualified"></xsd:attribute>
  ```

  Its associated business object attribute would have the following format:

  ```
  [Attribute]
  Name=ns2:Name
  Type=String
  AppSpecificInfo=attr_name=Name;type=attribute;attr_fd=qualified
  ...
  ```

  If an XML attribute does not specify the `form` attribute, the value of the `attributeFormDefault` attribute (on the `schema` element) determines whether attribute names are qualified. For more information, see "Qualified component names" on page 69.
- The `attr_ns` tag specifies the target namespace for the XML attribute, if this namespace is *different* from the schema document's target namespace. This tag is required when a schema document uses multiple namespaces. If the XML attribute referenced in one schema document is defined in the target namespace of *some other schema document*, this tag lists the name of that namespace.

  For example, suppose an XML attribute has the following definition:

```
<xsd:attribute ref="ns2:Name"></xsd:attribute>
```

Its associated business object attribute would have the following format:

```
[Attribute]
Name=Name
Type=String
AppSpecificInfo=attr_name=Name;attr_ns=http://www.example.com/ns2;
   type=attribute
...
```

For a complete example of a schema document that includes XML attributes defined in a second namespace, see "Schema namespaces" on page 64.

For an XML element or attribute that contains special characters: Business object attributes representing XML elements or XML attributes that include special characters in their content require escape processing by the XML data handler. To notify the data handler of the need to perform escape processing, the business object attribute's application-specific information must contain the following tag:

```
escape=true
```

The steps to specify escape processing for an XML document that uses a schema document to describe its schema are the same as to specify escape processing for an XML document that uses a DTD to describe its schema. For more information, see "For an XML element or attribute that contains special characters" on page 51.

For an XML comment: When the XML data handler converts a business object to an XML document, you can specify that it add the XML comments to the XML document by including the following tag in an attribute's application-specific information:

```
type=comment
```

The XML ODA does *not* automatically generate business object attributes for XML comments. You must manually add these attributes. The steps to add comments to an XML document described by a schema document are the same as to define a comment for an XML document that is described by a DTD. For more information, see "For an XML comment" on page 53.

For XML processing instructions: If the XML document contains XML processing instructions, some business object definition associated with the schema document must contain an attribute to hold the processing value. The application-specific information of this attribute must contain the following `type` tag:

```
type=pi
```

For example, when the XML data handler converts an XML document to a business object, it puts the XML version in a special attribute of the top-level business object definition called `XMLDeclaration`. The top-level business object in Figure 19 on page 57 shows the XMLDeclaration attribute within the TopLevel_Customer business object definition. For more information about the `type=pi` tag, see "For XML processing instructions" on page 53.

For XML schema locations: XML documents that refer to schema locations for their schema documents must include the following information:

- Declaration of the XML Schema Instance Namespace and a mapping of the `xsi` prefix to this namespace
- Inclusion of one of the following XML-Schema-instance attributes:
  - `xsi:schemaLocation`

This attribute associates the name of a namespace with a schema location. If the schema document uses a target namespace, the name of this target namespace must match the namespace that the `xsi:schemaLocation` attribute (in the XML document) specifies.

- `xsi:noNamespaceSchemaLocation`

  This attribute identifies a schema location. If the schema document does *not* use a target namespace, the XML document includes the `noNamespaceSchemaLocation` attribute to identify a single schema location.

For example, suppose the XML document has the namespace declaration in Figure 28.

```
<order xmlns="http://sampleDoc.org.ord"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.com/order order.xsd">
   ...
/<order>
```

Figure 28. Sample schema-location definition in an XML document

The schema document might have the following namespace declaration, which defines a target namespace:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns="http://www.example.com/order"
     targetNamespace="http://www.example.com/order">
```

For the business object that represents this XML document to be able to hold the information in the `schemaLocation` attribute, its business object definition must provide an attribute for this schema-location information. In the root-element business object definition, this schema-location information is represented as follows:

- For an XML document that uses the `schemaLocation` attribute, the top-level business object definition must contain an attribute with the following properties:
  - The attribute name is `schemaLocation` and this attribute is of type `String`.
  - The attribute must have the following `type` tag in its application-specific information:

    `type=xsischemalocation`
  - The value of this business object attribute is the value of the `schemaLocation` attribute.
- For an XML document that uses the `noNamespaceSchemaLocation` attribute, the root-element business object definition must contain an attribute with the following properties:
  - The attribute name is `noNamespaceSchemaLocation` and this attribute is of type `String`.
  - The attribute must have the following `type` tag in its application-specific information:

    `type=xsinoNSlocation`

The root-element business object definition for the schema document that represents this XML document includes the following attributes for the schema location:

```
[Attribute]
Name=schemaLocation
Type=String
AppSpecificInfo=type=xsischemalocation;
```

Note: When an attribute represents the `schemaLocation` or `noNamespaceSchemaLocation` XML attribute, it does *not* require the `type=attribute` tag in its application-specific information.

The XML ODA determines whether to generate a `schemaLocation` or `noNamespaceSchemaLocation` attribute in the top-level business object based on the value of its `DoctypeorSchemaLocation` ODA configuration property.

Creating business object definitions from schema documents

A schema document describes and constrains the content of an XML document. Therefore, the schema document is very useful for obtaining information needed for the business object definition. To translate the structure information in the schema document to a business object definition, you can use the XML Object Discovery Agent (ODA). For information on XML ODA, see "Using an XML ODA to create business object definitions" on page 80.

Supported schema-document structures

The XML ODA maps element declaration in a schema to an attribute in the business object definition. The type of the business-object attribute depends on the type specified in the XML element declaration. A simple type maps to a `String` Java type. Complex types map to a business object definition.

The XML ODA supports structures of a schema document including:

- Simple types — Simple types can be either atomic (built-in or derived using restriction), list, or union types. The XML ODA maps all these types to a business-object attribute type of string. For more information, see "For XML elements within a complex type" on page 73.
- Element wildcard`any` — The XML ODA maps the `any` wildcard to a simple attribute in the business object definition. At runtime, users must configure this attribute (by adding the appropriate application-specific information) based on the knowledge of the data.
- `anyAttribute` element — This element enables the user to extend the XML document with attributes that are not specified in the schema. The XML ODA prompts the user for a name for such an attribute and then maps the attribute to a simple attribute in a business object definition (see Figure 51 on page 220). The ODA ignores any `namespace` and `processContents` attributes for the `anyAttribute` wildcard.
- `sequence` groups — This model group requires child elements to appear in a specific order. The XML ODA maps the child elements in a `sequence` group to attributes within the business object definition. It determines the type of these attributes from the XML `type` attribute.
- `choice` groups — This model group indicates that only one of the corresponding conforming element declarations must appear in an instance. The XML ODA maps the child elements in a `choice` group to attributes within a wrapper business object definition. A parent business object definition contains a single attribute with multiple cardinality whose type is the wrapper business object definition. For more information, see "Wrapper business object definitions based on schema documents" on page 60.
- `all` groups — The XML ODA maps the child elements in an `all` group to attributes within a wrapper business object definition. A parent business object

definition contains a single attribute with multiple cardinality whose type is the wrapper business object definition. For more information, see "Wrapper business object definitions based on schema documents" on page 60.

- `include` element — The effect of the `include` is to bring into a schema document all the definitions and declarations contained in the included schema document. The XML ODA requires that you provide the full path to the schema document to be included. This full path is the actual location of the included schema document in the file system, *not* its URI. If you specify the location of the schema document with the XML ODA property `FileName`, all included schema documents must exist in this same location.

  Note: When you include a schema document, make sure that the included schema document does *not* declare global elements that have the same names as the schema document that does the `include`.
- `restriction` element — This element allows you to derive complex types by restriction; that is, to restrict a complex type by removing or restricting its attributes or content. The XML ODA supports the restriction element in the following contexts:
  - For simple-content restrictions, the XML ODA does not perform any mapping for the restrictions that are defined for simple content and that involve facets of the derived type. It does process restrictions on any attributes defined in the derived type.
  - For complex-content restrictions, the XML ODA supports the elimination or restriction of attributes and the content model of the base complex type, as follows:
    - For a restricted content model, the ODA creates a business object definition for *both* the base complex type and the derived type. However, the XML ODA assumes that the schema syntax is valid and follows the W3C Schema specification guidelines for restrictions; it does *not* validate if the derived types are valid restrictions of the base type.
    - For attribute restrictions, the XML ODA assumes that all attributes of the base type are passed down to the derived type. Therefore, the business object definition has attributes for all XML attributes defined in the restriction as well as any other attributes defined in the base type (that are not included in the restriction).
- `import` element — This element allows an XML document to refer to components from other namespace. By specifying the `elem_ns` or `attr_ns` tags in the attribute application-specific information, the business object definition can identify an imported namespace. For more information, see "Schema namespaces" on page 64.
- Complex-type derivation — The XML ODA supports derived complex types using restriction and extension. For these constructs, it creates a business object definition for the derived complex type based on its derived content model.
- Type substitution — The XML ODA property `TypeSubstitution` provides support for type substitution based on the xsi:type attribute in the schema. When the user sets `TypeSubstitution` to True in a drop down in the ODA, the XML ODA generates an additional wrapper object with the ASI ″typeSub=true″ and business objects representing the base type and its derived types as children of this wrapper object. For more information, see "Type substitution in business object definitions based on schema documents" on page 61.

Unsupported schema-document structures

The XML ODA is capable of processing most schema documents. However, it does *not* support the following schema-document structures:

- Element substitution — The XML data handler does *not* support element substitution at runtime. You must specify element substitution in the schema with the `substitutionGroup` attribute on a global element declaration.
- Redefinition of types and groups — The `redefine` element allows you to redefine a type (simple or complex) or a group (element or attribute) so that only certain behavior is supported.
- Default and fixed attributes — The XML ODA ignores default and fixed attributes on element and attribute declarations. If an element or attribute declaration specifies one of these attributes and the instance does *not* contain the element or attribute, the XML ODA does not populate the corresponding business-object attribute.

Creating business object definitions

An XML document can have either a DTD or a schema document to define its structure. The business object definitions that represent the elements in an XML document must contain information about the document's structure. To create business objects to be processed by the XML data handler, the XML data handler must be able to locate business object definitions that contain the structural information for each XML document to be processed. You can generate business object definitions for XML documents in one of the following ways:

- "Using an XML ODA to create business object definitions"
- "Manually creating business object definitions" on page 81

Both these techniques involve use of the Business Object Designer tool. This section provides an overview on how to use Business Object Designer to generate business object definitions for XML documents. For a complete description of Business Object Designer , refer to the *Business Object Development Guide*.

Note: Some connectors that use the XML data handler require a top-level wrapper business object that contains the content business objects as children. Depending on the business object structure required by the connector you are using, therefore, you may need to add generated business objects as children to another business object. See the documentation for each connector for information on the structure of its business objects.

Using an XML ODA to create business object definitions

The XML Object Discovery Agent (ODA) creates business object definitions for an XML document based on either its DTD or schema document. The ODA examines the DTD or schema document to obtain information about the XML document structure. It then writes the business object definitions to a file that can be loaded into the business integration system.

Note: If an XML document does *not* have a DTD or schema document, you can manually create a business object definition for the document. For more information, see "Manually creating business object definitions" on page 81.

The XML ODA builds business object definitions that conform to the requirements of the XML data handler. The ODA adds the required `ObjectEventId` attribute to all business object definitions. It also adds the repository version number to the top of the business object file if you specify this, which is required to import a business object definition into the InterChange Server business integration system. These

business object definitions do *not* usually need additional editing. However, if you need to perform edits, see "Modifying information in the business object definition" on page 221.

For information on how to use XML ODA, see "Using the XML ODA," on page 209. This appendix describes how to install and configure the XML ODA. It also describes how to use the XML ODA in Business Object Designer to generate business object definition. For information on launching Business Object Designer , see the *Business Object Development Guide*.

Manually creating business object definitions

This section describes how to manually create business object definitions to represent XML documents. You must ensure that you correctly define the business object definition, including its attributes, and the application-specific information.

Note: If an XML document does *not* have a DTD or schema document, you must manually create a business object definition for the document. If a DTD or schema document exists, IBM recommends use of the XML ODA to create a business object definition.

The description of XML-document format for a DTD or schema document describes business object definitions that the XML ODA builds. Table 11 on page 31 shows the sections of this manual that describe the format of XML documents that have a corresponding data model to describe their schema. As described in these sections, the business object definitions conform to the requirements of the XML data handler. Therefore, you can follow these descriptions when you need to create business object definitions manually.

In the following steps, *ElementTypeName* is the type of the XML element represented by the business object construct (either attribute or business object). To define a business object based on an XML document:

1. Create the top-level business object definition. The name of this business object definition should be the highest level element in the XML document (name of DTD or schema document, for example) in the format: *BOPrefix_TopLevelName*.

   Note: Some connectors require a wrapper business object that contains the content business object as children. Refer to "Wrapper business object definitions based on DTDs" on page 46 for more information.

2. In the top-level business object definition, create attributes for the XML elements.

   For a top-level business object definition based on either a DTD or a schema document, the following attributes are required:

   - `XMLDeclaration` – The attribute's application-specific information is `type=pi`.
   - An attribute to represent the root element in the DTD or schema document – The attribute contains a single-cardinality child business object and its application-specific information specifies the name of the element with the `elem_name` tag.

   For general information about these required attributes, see "Business object structure" on page 35. In addition, this document provides the following information about the structure of the top-level business object definition based on DTDs and schema documents:

Data model	For more information
Document type definition (DTD)	"Business object structure for DTDs" on page 43
Schema document	"Required business object definitions for schema documents" on page 56

3. Create the root-element business object definition, which is a child object of the top-level business object definition. It contains attributes for the root XML element. The name of this business object definition should be the root element in the XML document in the format: *BOPrefix_TopLevelName_RootElementName*
4. In the root-element business object definition, create a business object attribute for each contained element. Keep the following in mind:
   - The business object attribute name need not be the same as the XML element (or attribute) name. The application-specific information is used to specify the element (or attribute) name.
   - XML attributes must be the first attributes in the business object definition.
   - Type determination:
     - A String type is a cardinality-1 element with no element content or associated attribute-list declaration.
     - A business-object type is a cardinality-n contained element, or contained element with element content or associated attribute specification (or specifications).
   - Application-specific information is required for a String type attribute, mixed-typed element, or choice list element with cardinality n.

     For general information about this application-specific information, see "Application-specific information" on page 38. In addition, this document provides the following information about the application-specific of the business-object attributes based on DTDs and schema documents:

Data model	For more information
Document type definition (DTD)	"Application-specific information for XML components in DTDs" on page 45
Schema document	"Application-specific information for XML components in schema documents" on page 64

   Note: For an XML document with a schema definition, the root-element business object definition can also require an attribute for the schema location. For more information, see "Required business object definitions for schema documents" on page 56.
5. Create child business object definitions for all contained elements. Follow the rules listed above.

Converting business objects to XML documents

To convert a business object to an XML document, the XML data handler loops through the attributes in the business object definition in sequential order. It generates XML recursively based on the order in which attributes appear in the business object and its children.

The XML data handler processes business objects into an XML document as follows:

1. The data handler creates a document to contain the XML data.
2. The data handler examines the application-specific information in the top-level business object definition to determine if there are any child meta-objects (those whose names are listed in the `cw_mo_label` tag of the business-object-level application-specific information). The data handler does *not* include these attributes in the XML document.
3. The data handler loops through the remaining attributes of the business object definition.

   The data handler generates XML for each attribute using the following rules:

   - It creates a start and end tag for every simple `String` type attribute unless the `notag` string appears in the application-specific information for that attribute. If the start tag is open and the business object attribute being processed does *not* represent an XML attribute, the XML data handler closes the start tag (that is, it adds the character ">" to the XML document).
   - If the attribute represents a business object, the XML data handler opens a start tag, makes a recursive call to retrieve the attributes in the child business object, generates XML for the attributes of the child business object, and then generates an end tag for the element. The XML data handler uses the *attribute-level* application-specific information `elem_name` as the name of the element. For multiple-cardinality attributes, this process is repeated for each instance of the business object in the array.
   - If the attribute contains the application-specific information `xsiType`, the XML data handler writes out an attribute for the current element of the form `xsi:type=ValueofXsiType`. For example, if the business object attribute specifies `xsiType=ShirtType` then the corresponding XML attribute is `xsi:type="ShirtType"`. For more information about type substitution, see "Type substitution in business object definitions based on schema documents" on page 61.
   - For attributes that represent XML markup, the data handler uses the attribute application-specific information and generates XML as shown in Table 29..

Table 29. XML output for attributes that represent XML markup

XML entity business object attribute represents	XML output	Example	Application-specific information
Processing instruction	<?*AttrValue*?>	<?xml version="1.0"?>	type=pi
DTD	<!*AttrValue*>	<!DOCTYPE CUSTOMER "customer.dtd">	type=doctype
Element	<*ElementName*>... </*ElementName*>	For XML document based on a DTD: <CUSTOMER>... </CUSTOMER> For XML document based on a schema document: <element name=CUSTOMER...> </element>	type=pcdata
XML attribute	AttrName= "*AttrValue*"	For XML document based on a DTD: Seqno="1" For XML document based on a schema document: <element name=CUSTOMER... Seqno="1"...> </element>	type=attribute
CDATA section	<![CDATA[*AttrValue*]]>	<![CDATA [<HTML>Text</HTML>]]>	type=cdata

Table 29. XML output for attributes that represent XML markup (continued)

XML entity business object attribute represents	XML output	Example	Application-specific information
Comment	`<!--`*`CommentText`* `-->`	`<!--Customer information from source application A-->`	`type=comment`
Schema location (with target namespace)	`<`*`elementName`* `xmlns="`*`URI_path`*`" xmlns:xsi= "http://www.w3.org/ 2001/XMLSchema- instance" xsi:schemaLocation= "`*`URI_for_schema schema_location`*`" ...`	See Figure 28 on page 77	`type= xsischemalocation`
Schema location (no target namespace)	`<`*`elementName`* `xmlns="`*`URI_path`*`" xmlns:xsi= "http://www.w3.org/ 2001/XMLSchema- instance" xsi:noNamespace SchemaLocation= "`*`schema_location`*`" ...`	<order xmlns=″http://sampleDoc.org.ord″ xmlns:xsi=″http://www.w3.org/2001/ XMLSchema-instance″ xsi:noNamespaceSchemaLocation= ″order.xsd″> ... /<order>	`type=xsinoNSlocation`

- For attributes that contain the character data (`PCDATA` in a DTDs) associated with a given element, no markup is generated, and only the value of the attribute itself is added to the document (assuming the `notag` tag exists in the application-specific information).
- For attributes that contain character data for an element, or for an attribute, the data handler replaces any special characters with appropriate escape sequences (assuming the `escape=true` tag exists in the application-specific information), as Table 30 shows.

  Table 30. Special characters and the XML representations

Special character	XML escape sequences
ampersand (&)	`&amp`
less than (<)	`&lt`
greater than (>)	`&gt`
single quote (‘)	`&apos`
double quote (“)	`&quot`

Note: If an attribute represents an XML element whose value contains single quotes, double quotes or the &, <, or > characters, then the attribute requires escape processing. An attribute is *not* escape-processed unless it contains the value `escape=true` in its application-specific information.This application-specific information must be placed at the end of any other text.

- An attribute is skipped if either of the following conditions exist:
  - The attribute has the value of `CxIgnore`.
  - The attribute name is listed in the `cw_mo_` tag, in the application-specific information of the business object definition.

  No XML is generated for these attributes.

- Simple `String` attributes with a value of `CxBlank` are included in the XML document as empty tags. For XML documents based on DTDs, double quotation marks (" ") are used as the `PCDATA` equivalent of `CxBlank`. However, the data handler assumes that no complex attribute (one whose type is a business object) has a value of `CxBlank`.

4. When the data handler completes the conversion, it returns the XML document to the caller.

   Note: The verb information in a business object is lost in the conversion to an XML document. For information on preserving the verb, see "Business object verbs" on page 39.

Converting XML documents to business objects

This section provides the following information on how the XML data handler converts XML documents to business objects:

- "XML document requirements"
- "Serialized-data processing"

XML document requirements

The XML data handler makes the following assumptions about an XML document:

- The XML document is well formed.
- The XML document is compliant with SAX parser requirements. Note that the default parser used by the XML data handler requires that an XML document include an XML declaration.
- The structure of an XML document must match the structure of its corresponding business object. In addition, the order of the elements in the document must match the order of the attributes in the business object.

Serialized-data processing

When converting an XML document into a business object, the XML data handler assumes that the business object follows the structure of the XML document and conforms to the business object definition requirements described in "Requirements for business object definitions" on page 34. If there is no attribute in the business object for a given element name, the XML data handler returns an error.

To convert an XML document to a business object, the XML data handler does the following:

1. If the calling connector passes in a business object to the conversion method, the data handler uses this business object and continues. If the caller does *not* pass in a business object, the data handler determines the business object name and creates a business object to contain the data in the XML document.

   To determine the business object name, the data handler invokes the name handler. The default name handler forms the top-level business object name by combining the `BOPrefix` meta-object attribute, an underscore, and the value of the root element. For example, if the XML document contains `<!DOCTYPE Customer >` and the `BOPrefix` attribute is `MyApp`, the resulting name is `MyApp_Customer`. You can provide a custom name handler to configure different behavior.
2. The data handler retrieves the value of the `Parser` meta-object attribute to determine which SAX parser to use to parse the XML document. For

information on which SAX parser the XML data handler uses, see "SAX parser" on page 33. When the data handler determines the name of the parser, it instantiates the parser.

3. The data handler registers the event handler (for a DTD-based XML document, it also registers the entity resolver) with the parser. The event handler is a callback method that processes each XML element and attribute.

   Note: The entity resolver handles external entity references in DTD documents. If the data handler does not find an `EntityResolver` option with a valid class name, it uses `com.crossworlds.DataHandlers.xml.DefaultEntityResolver`. This entity resolver ignores all external references.

4. The data handler invokes the parser to parse the XML document.
   - The data handler determines if there are any child meta-objects (those whose names are listed in the `cw_mo_`*`label`* tag of the business object application-specific information). The data handler does *not* perform the processing to populate these attributes of the business object.
   - Depending on the type of element, the parser's event handler queries the business object definition for the attribute properties and processes element data accordingly. Execution is stopped only on fatal errors from the parser or when an element in the XML data cannot be found in the business object definition.
5. When the business object is complete, the data handler returns it to the caller.

Note: For every element and attribute found in an XML document, the data handler expects to find an attribute in the business object. If there is no attribute in the business object definition for a given element or attribute name, the data handler returns an error. The exception to this rule is for attributes of type FIXED, which are not required in a business object definition. If FIXED attributes are not present in the business object definition, execution does not stop if a FIXED attribute is found in the XML document.

Customizing the XML data handler

You can customize the XML data handler by:

- "Building a custom XML name handler"
- "Building a custom entity resolver" on page 88

Building a custom XML name handler

The XML data handler calls the name handler to extract the name of the business object from an XML message. The default name handler included with the XML data handler looks for the tag:

```
<!DOCTYPE Name>
```

From this tag and the `BOPrefix` meta-object attribute, the data handler generates the name of the business object. The XML data handler determines which name handler to invoke by using the value of the `NameHandlerClass` attribute stored in the data-handler meta-object. If you need the name handler to function in a different way, you must:

1. Create a custom name handler by extending the `NameHandler` class.

2. Configure the XML data handler to use the custom class by updating the default value of the NameHandlerClass attribute in the meta-object for the XML data handler.

The following sample code extends the DataHandler class to create a custom data handler, CustomDataHandler, for the XML data handler:

```
package com.crossworlds.DataHandlers.xml;

// DataHandler Dependencies
import com.crossworlds.DataHandlers.
     Exceptions.MalformedDataException;
import com.crossworlds.DataHandlers.NameHandler;
import com.crossworlds.DataHandlers.DataHandler;

// Java classes
import java.io.*;
import java.lang.Exception;

/**********************************************************
 * CustomNameHandler class. This class extends the Namehandler
 * class and implements method:
 *      getBOName( Reader serializedData, String subType )
 * The method getBOName contains the logic to extract the BOName
**********************************************************/
public class CustomNameHandler extends NameHandler
{

         /**
          * This method generates the business object name from
          * the data extracted from the 'serializedData' arg.
          * In this case, it is up to the caller to create
          * the BOName.
          */

         public String  getBOName( Reader serializedData,
                                   String subType )
           throws MalformedDataException
         {
           // The NameHandler uses DataHandler tracing. If the
           // DataHandler is not set, the NameHandler won't run.
           if (dh == null)
            return null;
           // Log a message
           dh.traceWrite(
           "Entering CustomNameHandler.getBOName for subtype '"
             + subType + "'.", 4);

           // This method parses the XML document and extracts the
           // business object name from the following tag in
           // the XML doc:
           //        <cml title=
           // For example, in:
           //        <cml title="cholestrol" id="cml_cholesterol">
           // the business object name is 'cholestrol'.

           // Log a message
           dh.traceWrite(
             "Name resolution will be done using <cml title= ",4);
           String name = null;

           try
           {
             // Read line of data from the Reader object
             LineNumberReader lineReader =
               new LineNumberReader( serializedData );
             serializedData.mark( 1000 );
```

```
                String line = lineReader.readLine();
                while ( line != null )
                {
                  // search for <cml title= in the line
                  int start = line.indexOf("<cml title=");
                  if ( start != -1 )
                  {
                    start += 12;
                    // search for the ending quotes for the tile tag
                    int end = line.indexOf('\"', start);

                    // extract name from line
                    name = line.substring(start, end);
                    break;
                  }
                  line = lineReader.readLine();
                }

                if ( name == null || name.length() == 0 )
                  throw new MalformedDataException(
                   "Error: can't determine the BusinessObject Name.");

              }

              catch(Exception e)
              {
                throw new MalformedDataException( e.getMessage() );
              }
              serializedData.reset();
              return name;
            }
}
```

Building a custom entity resolver

The SAX parser used by the XML data handler calls the entity resolver to find external entities (referenced DTDs) within an XML document. The entity resolver included with the XML data handler can ignore external references or search for them on a local file system. If you need to specify another way for external entities to be found, you must create a custom entity resolver class.

The XML data handler determines which entity resolver to invoke by using the value of the `EntityResolver` attribute stored in the XML data-handler meta-object.

Chapter 4. EDI data handler

The IBM WebSphere Business Integration Data Handler for EDI (Electronic Data Interchange), called the EDI data handler, converts business objects to EDI documents and from EDI documents to business objects. For instructions on installing the EDI data handler, see "Installing data handlers" on page 21.

This chapter describes how the EDI data handler processes EDI documents and how to define business objects to be processed by the data handler. You can use this information as a guide to implementing business objects that conform to the requirements of the EDI data handler. The chapter also discusses how to configure the XML data handler. This chapter contains the following sections:

Overview

The EDI data handler is a data-conversion module whose primary role is to convert business objects to and from EDI documents. An EDI document is a standardized format for conveying business information. The EDI data handler supports two message standards: X.12 and EDIFACT.

The EDI document is serialized data with the `edi` MIME type. The default top-level connector meta-object (`MO_DataHandler_Default`) supports the `edi` MIME type. Therefore, a connector that is configured to use the `MO_DataHandler_Default` data-handler meta-object can call the EDI data handler.

Attention

The EDI data handler is only capable of processing an EDI document containing a single group that contains only a single transaction type (the document may contain multiple transactions of the same type). Most EDI environments handle documents with multiple groups and transaction types. The IBM WebSphere Business Integration Adapter for TPI contains splitting logic that enables it to process EDI documents with multiple groups and transaction types. No other adapters contain this splitting logic so although they can technically use the EDI data handler, only the IBM WebSphere Business Integration Adapter for TPI is able to handle situations where the EDI document contains multiple groups.

The data handler parses document data using document separators that it identifies in the EDI document. If the data handler cannot identify the separators from the document, it uses separator values specified by attributes in the child meta-object associated with the EDI data handler. For more information on the EDI child meta-object, see "Configuring the EDI data handler child meta-object" on page 92.

EDI data-handler components

The EDI data handler uses a name handler to extract the name of the business object from an EDI message. Figure 29 illustrates the EDI data handler components and their relationship to one another.

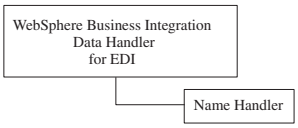

Figure 29. EDI data-handler components

The data handler invokes an instance of the name handler based on the value of the `NameHandlerClass` attribute in the EDI data handler child meta-object:

- If a class name is provided in the `NameHandlerClass` attribute, the EDI data handler uses this name handler to determine the business object name.

  The `NameHandlerClass` attribute in the version of the meta-object delivered with the product references the default EDI name handler, which obtains the name of the business object to create from the name-handler lookup file that the `NameHandlerFile` attribute specifies.
- If no class name is provided, the data handler logs an error and generates an exception.

For information on how to create a custom name handler, see "Customizing the EDI data handler" on page 112.

Business object and EDI document processing

The EDI data handler performs the operations listed in Table 31.

Table 31. Data operations for the EDI data handler

Data-handler operation	For more information
Receives a business object from the caller, converts the business object into an EDI document, and passes the EDI document to the caller.	"Converting business objects to EDI documents" on page 102
Receives an EDI document from the caller, builds a business object, and returns the business object to the caller.	"Converting EDI documents to business objects" on page 105

Configuring the EDI data handler

To configure the EDI data handler for use with a connector, take the following steps:

- Create an EDI name-handler lookup file for transaction identifiers, DUNS numbers, and business object names.
- Enter the appropriate values for the attributes of the EDI child meta-object.

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

Note: To use the EDI data handler, you must also create or modify business object definitions so that they support the data handler. For more information, see "Business object definitions for EDI documents" on page 94.

Creating the name-handler lookup file

The EDI data handler relies on a EDI name-handler lookup file to determine which business object to create based on the EDI message. This name-handler lookup file contains the following tab-separated columns:

- The transaction identifier (ID): this value identifies the EDI document type (for example, 850). It does not uniquely identify the contents of the EDI document. However, it does indicate the kind of information that the document contains.
- (Optional) The version number: this is the Version/Release/Industry Identifier Code that the EDI data handler uses to manage multiple versions of EDI documents
- The DUNS number: the EDI data handler uses this number to uniquely identify a trading partner.
- The name of the associated business object: the EDI data handler uses this business object name to identify the top-level EDI business object that it must create.

Note: Separate the fields of the name-handler lookup file with tab characters.

A sample of a name-handler lookup file without the optional version number follows:

```
850             123465      X12_850A_Order
850             122227      X12_850B_Order

855             122227      X12_855A_Response
855             123465      X12_855A_Response
```

A sample of a name-handler lookup file with the version number follows (the version numbers appear in the second column; in this example the version number is `004010`):

```
850             004010    111111      X12_850A_Order
855             004010    122227      X12_855A_Response
```

Note: In this example, "`X12_`" is used as a common prefix for the top-level business object names. This prefix is *not* required. You choose an identifying prefix when you create your top-level business objects. For more information, see "Top-level EDI business object" on page 95.

To provide the EDI data handler with information about the business objects it creates, you must:

- Ensure that there is an entry in the name-handler lookup file for each combination of DUNS number and transaction ID (and optionally the version number) for which the data handler is to create a business object. Make sure column values are separated by tab characters.
- Set the `NameHandlerFile` meta-object attribute to the fully qualified pathname for this name-handler lookup file.

Note: The Default Value property of the `NameHandlerFile` attribute in the delivered version of the child meta-object contains a value. You must ensure, however, that the pathname in the `NameHandlerFile` attribute specifies the name of your EDI name-handler lookup file. When you specify the path on a Windows system, you must escape all backslash (`\`) characters by including a second backslash. For example:

```
c:\\home\\DataHandlers\\edi\\edi_xref
```

UNIX pathnames do not use a backslash and therefore do not need to be escaped:

```
/home/DataHandlers/edi/edi_xref
```

The EDI data handler refreshes the information from this file each time the file is updated. Therefore, it picks up new or changed values immediately, so you do not have to restart any components.

Configuring the EDI data handler child meta-object

To configure an EDI data handler, you must ensure that its configuration information is provided in the EDI data handler's child meta-object. For the EDI data handler, IBM delivers the default child meta-object `MO_DataHandler_DefaultEDIConfig`. Each attribute in this meta-object defines a configuration property for the EDI data handler. Table 32 describes the attributes in this child meta-object.

Table 32. Child meta-object attributes for the EDI data handler

Attribute name	Description	Delivered default value
`ClassName`	Name of the data handler class to load for use with the specified MIME type. The top-level data-handler meta-object has an attribute whose name matches the specified MIME type and whose type is the EDI child meta-object (described by Table 32).	`com.crossworlds. DataHandlers.edi. edi`
`DefaultVerb`	Name of the verb to set in the business object when converting from an EDI document to a business object. If no value exists for this attribute, the EDI data handler does not include a verb in the business object.	`Create`
`DummyKey`	Key attribute; not used by the data handler but required by the business integration system.	1
`ISA` (X.12 standard) `UNA` and `UNB` (EDIFACT standard)	Provides positional information for separators so that the EDI data handler can obtain the values of separators from the EDI document itself. The name of this attribute must correspond with the name of the first segment in your EDI document, as follows: • If your EDI messages follow the X.12 standard, the EDI document starts with a segment named `ISA`; this positional-information attribute is named `ISA`. • If your EDI messages follow the EDIFACT standard, the EDI document starts with a `UNA` service string advice (optional) and an initial segment named `UNB`. Therefore, you must create two positional-information attributes in this meta-object: `UNA` and `UNB`. For information about the values in this meta-object attribute, see 106.	*None*
`NameHandlerClass`	Name of the class to use to determine the name of a business object from the content of an EDI document. Change the Default Value property of this attribute if you create your own custom name handler. For more information, see "Customizing the EDI data handler" on page 112.	`com.crossworlds. DataHandlers.edi. EdiNameHandler`

Table 32. Child meta-object attributes for the EDI data handler (continued)

Attribute name	Description	Delivered default value
NameHandlerFile	Fully qualified name of the EDI name-handler lookup file, which contains a name-handler lookup table for transaction IDs, an optional version number, DUNS numbers, and business object names. For more information, see "Creating the name-handler lookup file" on page 91.	Windows systems: C:\\crossworlds\edi\dbfile.txtUNIX systems: /home/crossworlds/edi/dbfile.txt
RELEASE_CHAR	The character to use as an escape character in the attribute value. This escape character is necessary if any of the EDI document separators is part of the actual value of an attribute. You must precede the character in the actual value with this escape character. For example, if an attribute value is "*dog?" and the element separator is the asterisk, you must escape the asterisk in the attribute value, as follows: "?*dog??".	? (question mark)
SEPARATOR_ELEMENT	The character used as the element separator in the EDI document.	* (asterisk)
SEPARATOR_COMPOSIT	The character used as the composite separator in the EDI document.	, (comma)
SEPARATOR_REPEAT	The character used as the repeat separator in the EDI document. It is used to separate repeating composites.	^ (caret)
SEPARATOR_SEGMENT	The character used as the segment separator in the EDI document. If you want to set the segment separator to a newline character, you must escape the character, as follows: • On Windows systems – X12 documents: \r\n – EDIFACT documents: \n • On UNIX systems: \n	~ (tilde)
ObjectEventId	Not used by the data handler but required by the business integration system.	*None*

The "Delivered default value" column in Table 32 lists the value in the Default Value property for the corresponding attribute in the delivered business object. You must examine your environment and set the Default Value properties of all the attributes to the appropriate values.

Note: Use Business Object Designer to modify business object definitions.

To invoke multiple configurations of the EDI data handler, take the following steps:

- Copy and rename the default EDI child meta-object (which configures the EDI data handler for EDI documents in the X.12 standard). A recommended approach to naming a new child meta-object is to provide subtypes to the MIME type. For example, you can rename the default EDI child meta-object to MO_DataHandler_DefaultEDI_X12Config and name its copy MO_DataHandler_DefaultEDI_EDIFACTConfig.
- Set the default values of the attributes in each EDI child meta-object to configure the data handler instance.
- Create attributes in the top-level data-handler meta-object named edi_*subtype*, where *subtype* can be one of the EDI standards. To handle EDI documents in

either X.12 or EDIFACT standard, you can create two attributes in the top-level meta-object: `edi_x12`, and `edi_edifact`. Each of these attributes would represent its associated child meta-object.

For more information about how to configure a data handler, see "Configuring data handlers" on page 24.

Business object definitions for EDI documents

To use the EDI data handler, you must create or modify business object definitions so that they contain the metadata that the data handler requires and so that they include the fields that correspond to those in the EDI message. This section provides the information you need to create business object definitions to work with the EDI data handler. In particular, it provides the following information:

- "Understanding EDI business object structure"
- "Creating business object definitions for EDI documents" on page 101

Understanding EDI business object structure

The EDI data handler uses business object definitions when it converts business objects or EDI documents. It performs the conversion using the structure of the business objects and their application-specific information. Figure 30 shows the structure of the business objects that represent an EDI message.

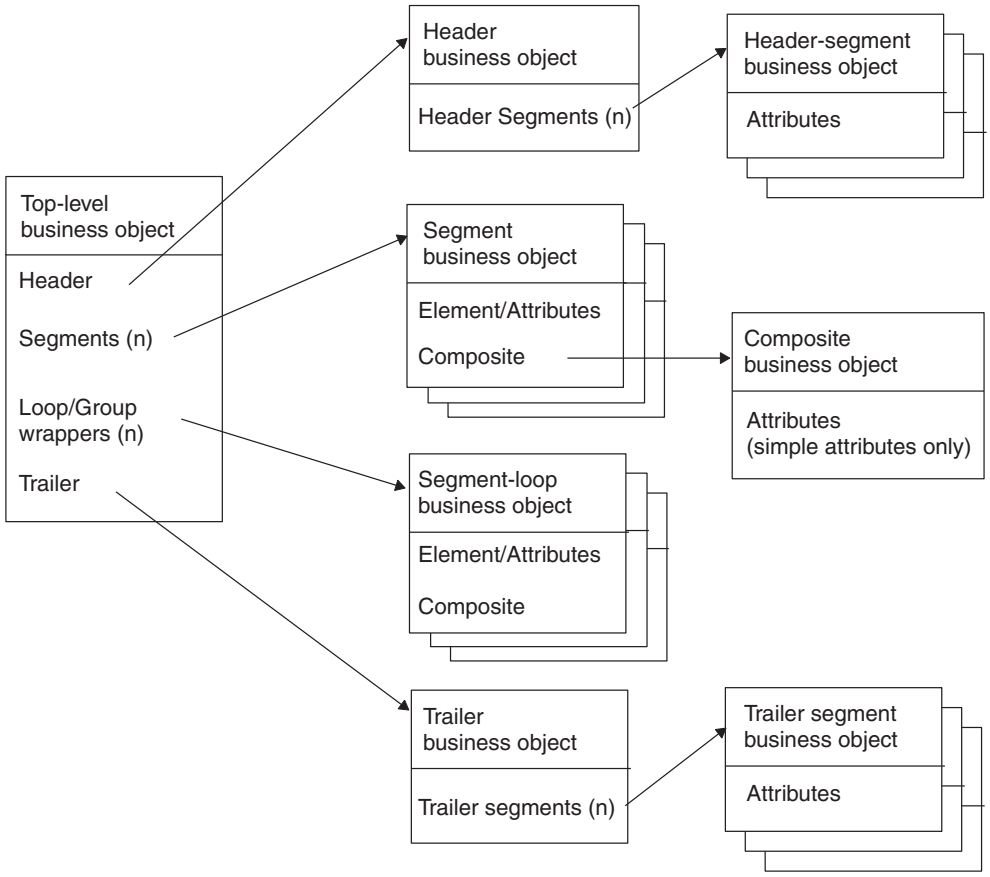

Figure 30. Business object structure for an EDI message

To ensure that business object definitions conform to the requirements of the EDI data handler, use the guidelines provided for each of the following business objects:

- "Top-level EDI business object"
- "Header business object" on page 97
- "Segment business object" on page 98
- "Composite business object" on page 99
- "Segment-loop business object" on page 100
- "Trailer business object" on page 100

Top-level EDI business object

The EDI data handler expects a top-level business object to hold the information for the EDI message. Table 33 describes how the EDI data handler interprets the properties of a business object and describes how to set the properties when modifying a business object for use with the EDI data handler.

Table 33. Properties for the EDI top-level business object definition

Property name	Description
Name	Each business object definition must have a unique name. It is recommended that these business object definitions begin with a standard prefix. The name of the top-level business object depends on the message standard, as follows: • For an EDI message that follows the EDIFACT standard, the business object name has the form: *BusObj Prefix + Message Type* • For an EDI message that follows the X.12 standard, the business object name has the form: *BusObj Prefix + Transaction Set Identifier Code*
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific information	No metadata that includes a `type` tag. Might have metadata with `cw_mo` tags to indicate attributes that are to be ignored during the conversion process.

Note: When the data handler converts an EDI document to a business object, it identifies the top-level business object for the EDI document through the name-handler lookup table. For more information, see "Creating the name-handler lookup file" on page 91.

As Figure 30 shows, the top-level business object definition contains the following attributes:

- An attribute to represent the EDI document header
- As many attributes as needed to represent any segments
- As many attributes as needed to represent any segment loops or groups
- An attribute to represent the EDI document trailer

Header attribute: The header attribute of the top-level EDI business object represents a single-cardinality array that contains the header information. The application-specific information for this attribute must include the following tag:

```
type=header
```

The Type property for this attribute contains the name of the header business object. For information on the attributes of the header business object, see "Header business object" on page 97.

Segment attributes: Each segment attribute of the top-level EDI business object represents a single-cardinality array that contains the segment information. Table 34 shows the attribute properties for a segment attribute in the top-level business object definition.

Table 34. Attribute properties for a segment attribute in the EDI top-level business object definition

Property name	Description
Name	The name of a segment attribute takes the form: *Tag + Position* (if duplicated)
Type	The type of a segment attribute takes a name of the form: *TopLevelBusObj + Tag* This attribute property contains the name of the appropriate segment business object.
ContainedObjectVersion	A constant representing the current version of the business object definition. Current value is 1.0.0.
Relationship	Set to "containment".
Cardinality	If Max Use or Repetition is set to "1" in the EDI documentation specification, set the value of this attribute property to "1". Otherwise, set this attribute property to "N".
MaxLength	Always set to "1".
Key	Always set to "false".
Foreign key	Always set to "false".
Required	If Status or Option is set to "M" in the EDI documentation specification, set this attribute property to "true". Otherwise, set this attribute property to "false".
Default value	Not used by the EDI data handler.
Application-specific information	Set to: name=*name of segment*

Note: The Max Use, Repetition, Status, and Option fields are part of the EDI document specification. For more information, refer to your EDI documentation.

For information on the attributes of the segment business object, see "Segment business object" on page 98.

Segment-loop attributes: Each segment-loop attribute represents a multiple cardinality array that contains the segment information. Table 35 shows the attribute properties for a segment-loop attribute in the top-level business object definition.

Table 35. Attribute properties for a segment-loop attribute in the EDI top-level business object definition

Property name	Description
Name	The name of a segment-loop attribute takes the form: *Tag + Position* (if duplicated)

Table 35. Attribute properties for a segment-loop attribute in the EDI top-level business object definition (continued)

Property name	Description
Type	The type of a segment loop attribute takes a name of the form: *TopLevelBusObj + Loop/Group keyword + Tag* This attribute property contains the name of the appropriate segment-loop business object. This name can include a keyword (such as Loop) to indicate the purpose of the segment.
ContainedObjectVersion	A constant representing the current version of the business object definition. Current value is 1.0.0.
Relationship	Set to "containment".
Cardinality	Always set to "N".
MaxLength	Always set to "1".
Key	Always set to "false".
Foreign key	Always set to "false".
Required	If Status or Option is set to "M" in the EDI documentation specification, set this attribute property to "true". Otherwise, set this attribute property to "false". Only set this attribute property to "true" on the first segment in the Loop/Group.
Default value	Not used by the EDI data handler.
Application-specific information	Includes: • name=*name of first segment in loop* • type=loop For example, the following application-specific information identifies a segment loop whose first segment name is AMT: AppSpecificInfo=name=AMT;type=loop

Note: The Status and Option fields are part of the EDI document specification. For more information, refer to your EDI documentation.

For information on the attributes of the segment-loop business object, see "Segment-loop business object" on page 100.

Trailer attribute: The trailer attribute of the top-level EDI business object represents a single-cardinality array that contains the trailer information. The application-specific information for this attribute must include the following tag:

```
type=trailer
```

The Type property of this attribute contains the name of the trailer business object. For information on the attributes of the trailer business object, see "Trailer business object" on page 100.

Header business object

To hold header information for the EDI message, the EDI data handler expects a header business object as the first attribute of the top-level business object. Table 36 describes how the EDI data handler interprets the properties of this business object definition and describes how to set these properties when modifying the business object for use with the EDI data handler.

Table 36. Properties for the EDI header business object definition

Property name	Description
Name	Each business object definition must have a unique name. It is recommended that the name of the header business object include the business object prefix. It can also include identifying information such as the keyword "header".
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific information	No metadata that includes a type tag. Might have metadata with cw_mo tags to indicate attributes that are to be ignored during the conversion process.

This business object definition contains an attribute to represent each header segment of the header. The application-specific information for each attribute identifies the name of the header segment. Each header segments can contain attributes that are simple, single-cardinality, or multiple cardinality.

Segment business object

To hold segment information for the EDI message, the EDI data handler expects a segment business object. Table 37 describes how the EDI data handler interprets the properties of this business object definition and describes how to set these properties when modifying the business object for use with the EDI data handler.

Table 37. Properties for the EDI segment business object definition

Property name	Description
Name	Each business object definition must have a unique name. It is recommended that the name of the segment business object definition have the form: *BusObj Prefix + Tag*
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific information	No metadata that includes a type tag. Might have metadata with cw_mo tags to indicate attributes that are to be ignored during the conversion process.

As Figure 30 shows, the segment business object can contain the following attributes:

- A simple (String) attribute to represent an EDI element
- An array attribute to represent a composite

Simple attribute: Each simple attribute of a segment business object must have the attribute properties shown in Table 38.

Table 38. Attribute properties for simple attributes

Property name	Description
Name	Each business object attribute must have a unique name.
Type	Each simple business object attribute must have a String type.
Cardinality	Always set to "1".
Key	Used for simple attributes only: must be set for the *first* string attribute of the business object.
MaxLength	Set to the maximum size of this String attribute. Within an EDI document, when you embed a separator character as part of actual data

Table 38. Attribute properties for simple attributes (continued)

Property name	Description
Foreign key	Always set to "false".
Required	If Status or Option is set to "M" in the EDI documentation specification, set this attribute property to "true". Otherwise, set this attribute property to "false".
Default value	Not used by the EDI data handler.

Note: The Repetition, Status, and Option fields are part of the EDI document specification. For more information, refer to your EDI documentation.

Composite attribute: Each composite business object is an array that contains the elements of an EDI composite. Table 39 shows the attribute properties for a composite attribute.

Table 39. Attribute properties for a composite attribute in an EDI segment business object definition

Property name	Description
Name	The name of a composite attribute takes the form:*Tag + Position* (if duplicated)
Type	The type of a segment attribute takes a name of the form: *BusObj Prefix + Tag* This attribute property contains the name of the appropriate composite business object.
ContainedObjectVersion	A constant representing the current version of the business object definition. Current value is 1.0.0.
Relationship	Set to "containment".
Cardinality	If Repetition is set to 1, set the value of this attribute property to "1". Otherwise, set this attribute property to "N".
MaxLength	Always set to "1".
Key	Always set to "false".
Foreign key	Always set to "false".
Required	If Status or Option is set to "M", set this attribute property to "true". Otherwise, set this attribute property to "false".
Default value	Not used by the EDI data handler.
Application-specific information	*None*
Required Server Bound	Always set to "false".

Note: The Repetition, Status, and Option fields are part of the EDI document specification. For more information, refer to your EDI documentation.

For more information, see "Composite business object."

Composite business object

To hold composite information for an element in the EDI message, the EDI data handler expects a composite business object.

Note: Composites are usually found in EDI documents that follow the EDIFACT standard. However, they can exist in documents that follow the X.12 standard as well.

Table 40 describes how the EDI data handler interprets the properties of this business object definition and describes how to set these properties when modifying the business object for use with the EDI data handler.

Table 40. Properties for the EDI composite business object definition

Property name	Description
Name	Each business object definition must have a unique name. It is recommended that the name of the composite business object definition have the form: *BusObj Prefix + Tag*
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific information	No metadata that includes a `type` tag. Might have metadata with `cw_mo` tags to indicate attributes that are to be ignored during the conversion process.

The composite business object can contain simple (`String`) attributes or arrays.

Segment-loop business object

To hold information for a segment loop or group in the EDI message, the EDI data handler expects a segment-loop business object. Table 41 describes how the EDI data handler interprets the properties of this business object definition and describes how to set these properties when modifying the business object for use with the EDI data handler.

Table 41. Properties for the EDI segment-loop business object definition

Property name	Description
Name	Each business object definition must have a unique name. It is recommended that the name of the segment-loop business object definition have the form: *BusObj Prefix + Tag*
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific information	No metadata that includes a `type` tag. Might have metadata with `cw_mo` tags to indicate attributes that are to be ignored during the conversion process.

Trailer business object

To hold trailer information for the EDI message, the EDI data handler expects a trailer business object. Table 42 describes how the EDI data handler interprets the properties of this business object definition and describes how to set these properties when modifying the business object for use with the EDI data handler.

Table 42. Properties for the EDI trailer business object definition

Property Name	Description
Name	Each business object definition must have a unique name. It is recommended that the name of the trailer business object include the business object prefix. It can also include identifying information such as the keyword "`trailer`".
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific information	No metadata that includes a `type` tag. Might have metadata with `cw_mo` tags to indicate attributes that are to be ignored during the conversion process.

This business object definition contains an attribute to represent each trailer segment of the trailer. The application-specific information for each attribute identifies the name of the trailer segment. The trailer segments can contain simple, single-cardinality, or multiple cardinality attributes.

Creating business object definitions for EDI documents

There are two ways to create business object definitions for an EDI document:

- You can use Edifec SpecBuilder to export the definition EDI document as a business object definition.
- You can manually create a business object definition for the document.

Using SpecBuilder to create business object definitions

SpecBuilder can function as an object discovery utility, creating business object definitions based on an EDI document. It writes them to a business object definition file that can then be loaded into the business integration system. SpecBuilder is a third-party application released and supported by Edifecs Inc. Please consult the SpecBuilder documentation or the Edifecs web site for assistance.

Note: IBM does not include the SpecBuilder tool as part of its release. The tool is, however, available on an Edifecs CD. To obtain a copy of the Edifecs CD, please contact your IBM account executive or technical support.

Manually creating business object definitions

This section describes how to manually create business object definitions to represent EDI documents. Use Business Object Designer to add or delete attributes from the business object definition as well as edit attribute properties, as needed.

Note: The structure of an EDI document can be quite complex. It is recommended that you use SpecBuilder to build as much of a business object definition as possible.

To define a business object based on an EDI document:

1. Create the top-level business object definition.

   For information on the structure of this top-level business object, see "Top-level EDI business object" on page 95.
2. Create the child business objects for the top-level business object. In the top-level business object, create a child object attribute for the business objects shown in Table 43.

Table 43. Business objects for the EDI data handler

Part of EDI document	Notes	Business object
Header	This header might also contain header segments.	"Header business object" on page 97
Segments	A segment might also contain composites	"Segment business object" on page 98, "Composite business object" on page 99
Segment Loops and Groups	A segment loop consists of repeated segments.	"Segment-loop business object" on page 100
Trailer	This trailer might also contain trailer segments.	"Trailer business object" on page 100

Keep the following in mind:

- The business object attribute name need not be the same as the EDI element name. The application-specific information is used to specify the element name.

- Type determination: `String` is a cardinality 1 contained element with no element content or associated attribute-list declaration. `BusinessObject` is a cardinality `n` contained element, or contained element with element content or associated attribute specification(s).
- Application-specific information is required for many of the attributes. See the information in "Top-level EDI business object" on page 95 for more information.

3. Create a business object attribute for each simple element. For more information, see "Simple attribute" on page 98.
4. Create child business objects for any nested business objects, such as header segments, trailer segments, and composites. Follow the rules listed above.

Converting business objects to EDI documents

To convert a business object to an EDI document, the EDI data handler loops through the attributes of the top-level business object definition. It processes the attributes recursively, in the order in which they appear in the top-level business object, writing attribute values as the elements of the EDI document.

The EDI data handler processes business objects into an EDI document as follows:

1. The data handler initializes itself by setting the composite, element, segment, and repeat separators based on the configuration information in the child meta-object. If no value is provided for one of these configuration options, the data handler uses hard-coded defaults, as Table 44 and Table 47 show.

Table 44. Default values for element and segment separators

Precedence step		Element separator	Segment separator
1	Obtain the value of the corresponding meta-object attribute.	`SEPARATOR_ELEMENT`	`SEPARATOR_SEGMENT`
2	If the associated meta-object attribute is not set, use a hard-coded default.	plus sign (+)	single quote (')

2. The data handler examines the application-specific information in the top-level business object definition to determine if there are any child meta-objects (those whose names are listed in the `cw_mo_` tag of the business object application-specific information). The data handler does *not* include these attributes in the EDI document. For more information about the `cw_mo_` tag, see "Implementing conversion from a business object" on page 177.
3. The data handler loops through the remaining attributes in the top-level business object definition. Based on the cardinality of each attribute, the data handler determines what part of the EDI document the attribute represents. For more information, see "Determining the EDI data associated with the attribute" on page 103.
4. Once the data handler identifies the associated EDI data, it can take the appropriate steps to write the attribute data to the EDI document:
   - If an attribute represents a segment, the data handler checks whether it is null. If the business object is null, the data handler skips the attribute. If the business object is *not* null, the data handler performs the segment-processing steps. For more information, see "Processing a segment" on page 104.
   - For each child business object in a segment that represents a composite, the data handler loops through the attributes (all of which should be `String`), and takes the composite-processing steps For more information, see "Processing a composite" on page 104.

5. The data handler writes the total number of segments that it has written to this document into the "number of segments" field of the document. The data handler determines the position of this field from the `seg_count` tag in the `ISA` attribute of the child meta-object. This field is often located in the `SE` segment. For information on setting the `seg_count` tag, see "Obtaining positional information" on page 106.
6. When the data handler completes the conversion, it returns the serialized data to the caller. The data handler returns the data as a string that contains the EDI document.

Determining document separators to insert

To convert a business object to an EDI document, the EDI data handler must correctly insert separators into the EDI document. The data handler uses the attributes in the child meta-object to determine the values to assign these separators. If no value is provided for one of these attributes, the data handler uses hard-coded defaults for the separator.

Table 45 shows the EDI separators with their corresponding meta-object attributes and hard-coded default values.

Table 45. Default values for EDI separators

EDI separator	Separator attribute in child meta-object	Hard-coded default
Element separator	`SEPARATOR_ELEMENT`	plus sign (+)
Segment separator	`SEPARATOR_SEGMENT`	single quote (')
Composite separator	`SEPARATOR_COMPOSIT`	colon (:)
Repeat separator (EDIFACT documents only)	`SEPARATOR_REPEAT`	caret (^)

Attention:

EDI documents that follow the X.12 standard*:* For the data handler to properly convert business objects to an EDI document, the values of the separator attributes in the child meta-object *must* match those in the `ISA` segment of the EDI document. The EDI data handler does *not* read data in the `ISA` segment to determine document separators.

EDI documents that follow the EDIFACT standard: For the data handler to properly convert business objects to an EDI document, the values of the separator attributes in the child meta-object must be set to the default values, as defined in Table 45. The separator attributes in the default child meta-object (`MO_DataHandler_DefaultEDIConfig`) contain values valid for the X.12 standard. Make sure you reset these attribute default values to the values defined in Table 45.

Determining the EDI data associated with the attribute

The structure of the business objects that hold EDI is determined by the EDI document specification. (For information on how to create this business object structure, see "Creating business object definitions for EDI documents" on page 101.) The EDI data handler uses the cardinality of the attribute to determine what part of the EDI document this attribute represents. Based on this cardinality, the data handler takes the following actions:

- If the attribute represents a *single-cardinality array*, the data handler examines the application-specific information of the attribute to determine the part of the EDI document associated with this attribute.

The data handler checks the application-specific information of the attribute for a `type` tag (such as `type=header` or `type=trailer`) that identifies its purpose in the EDI structure.

  - If the data handler finds a `type` tag, it recursively processes the child business object.
  - If the data handler does *not* find such a tag, the data handler assumes that the child business object represents a segment in the EDI document and processes as described in "Processing a segment."

- If the attribute represents a *multiple cardinality array*, it represents a segment loop. The data handler recursively processes each child business object as if it were a single-cardinality array. It writes a new segment in the EDI document for each instance of a business object in the array.
- If the attribute is of type `String`, the data handler generates an exception because the EDI structure dictates that all attributes of the top-level business object are either single-cardinality or multiple cardinality arrays.

Processing a segment

If an attribute represents a segment, the actions that the data handler takes depend on whether the attribute is null:

- If the attribute value is null, the data handler skips the attribute; it does *not* include it in the EDI document.
- If the attribute value is not null, the data handler performs the following processing steps:
  - Parse the business object application-specific information for the segment name, a tag of the form:

    `name=`*`segment_name`*
  - Append the segment separator to the EDI document.
  - Append the segment name to the EDI document, insert any escape characters needed.
  - For each child business object (either single-cardinality or multiple cardinality), the data handler appends the element separator to the EDI document and processes the child business object as a composite (see "Processing a composite"). For a multiple cardinality attribute, the data handler processes each child business object in the order it occurs.
  - For each `String` attribute that is *not* null, the data handler appends the element separator and the value of the attribute, inserting any escape characters as needed.

Processing a composite

For each child business object that represents a composite, the data handler loops through the attributes (all of which should be `String`), and takes the following processing steps:

- Parse the attribute data, adding any necessary escape characters.
- Append the attribute value with any escape characters to the EDI document.
- Append the composite separator to the document.

Converting EDI documents to business objects

To convert an EDI document to a business object, the EDI data handler loops through the attributes of the top-level business object definition. It obtains the name of the business object to create, then processes the attributes recursively, in the order in which attributes appear in the top-level business object and its children, assigning element values from the EDI document to the business object.

The EDI data handler processes an EDI document into a business object as follows:

1. The data handler sets any properties that were passed in through the optional configuration object. This information would be passed in through the `config` argument of the `getBO()` method.
2. The data handler initializes itself to prepare for reading the EDI document. For more information, see "Initializing the data handler."
3. If the data handler does not receive a business object from the caller, it must create one based on the business object name it finds in the name-handler lookup file. For more information, see "Determining the name of the business object" on page 110.
4. Once the data handler has access to an instance of the top-level business object, the data handler populates this business object and its children with data from the EDI document. For more information, see "Populating the business object" on page 111.
5. When the data handler completes the conversion, it returns the top-level business object to the caller. The data handler returns the entire hierarchy, the top-level business object and all its child objects.

Initializing the data handler

To initialize itself to convert an EDI document to a business object, the EDI data handler takes the following steps:

1. Check that the `Reader` object that contains the serialized data supports the `mark()` operation.
2. Begin parsing the EDI document to obtain the first segment name, the separators, the transaction ID, and DUNS number.

Each of these steps is described in more detail in the subsections that follow.

Checking the Reader object

The EDI data handler must be able to mark a particular position within the EDI document and then subsequently return to that position. Because the EDI document is passed to the EDI data handler as a `Reader` object, this `Reader` object must be able to support the `mark()` operation.

As its first initialization step, therefore, the EDI data handler checks that the `Reader` object that it receives supports the `mark()` operation. If not, the data handler logs an error and generates an exception. It is recommended that all serialized data be passed into the EDI data handler in a `StringReader` object.

Note: For more information about a `Reader` object and the `mark()` operation, see the Notes section in the description of "getBO() - public" on page 198.

Determining the document separators to read

To convert an EDI document to a business object, the EDI data handler must correctly read separators in the EDI document. The data handler parses the document to obtain these separators. Because the first three characters of the EDI

document are known, the data handler parses these characters first. It reads the first three characters to determine if they represent:

- The UNA service string advice (EDIFACT documents only)

  The UNA service string advice contains the document separators to use.
- The initial segment name

  The data handler must parse the document to obtain the document separators and other positional information.

Checking for the UNA service string advice: The UNA service string advice is the optional first element in EDI documents that follow the EDIFACT standard. This service string consists of six alphanumeric characters in the following order:

Component separator
Element separator
Decimal mark
Release character
Repeat separator (syntax version 4 only)
Segment separator

If the first 3 characters of the EDI document are "UNA", the data handler uses the values that the UNA service string specifies to interpret the EDI document. These separator values take precedence over any other separator settings in the EDI document, including any in the UNA or UNB positional-information attributes of the child meta-object.

Note: For an EDI document with a UNA service string advice, the data handler obtains the transaction ID and the DUNS number from the UNA positional-information attribute of the child meta-object. For more information, see the next section.

Obtaining positional information: If the first 3 characters of the EDI document are *not* "UNA", the data handler assumes that they represent the name of the initial segment. The data handler assumes that the initial segments are part of the header and have names that are exactly three characters in length. If no UNA service string advice exists, the data handler must obtain document separators from the EDI document itself. The data handler continues parsing the EDI document, performing the following tasks:

- Read the fourth character of the EDI document to determine the element separator.

  If the data handler is unable to determine the element separator, it uses the value of the SEPARATOR_ELEMENT attribute from the child meta-object. The delivered value for SEPARATOR_ELEMENT is an asterisk (*). If, for any reason, the data handler is unable to obtain the element separator from the child meta-object, it uses its hard-coded default of a plus sign (+).
- Obtain the positional information from the attribute in the child meta-object

  Positional information includes the separators (segment, composite, and repeat separators), transaction ID, and DUNS number. The data handler can usually determine this positional information by scanning the EDI document.

  To help the data handler locate the positional information within the EDI document, the child meta-object associated with the EDI data handler (MO_DataHandler_DefaultEDIConfig by default) contains an attribute that holds this positional information. The name of this positional-information attribute corresponds to the name of the first element in the EDI document, as follows:

– For the X.12 standard, the EDI document starts with a segment named "ISA". Therefore, the data handler looks for an ISA attribute in the child meta-object.
– For the EDIFACT standard, most EDI document starts with a segment named "UNB". However, an optional UNA service string advice might occur first. Therefore, the data handler looks for an attribute in the child meta-object that matches the initial element (UNA, UNB, or whatever the name of the first element is). The data handler has no way of knowing whether a UNA service string advice exists until it begins parsing the EDI document. Therefore, both the UNA and UNB attributes must exist in the child meta-object.

Note: By default, the MO_DataHandler_DefaultEDIConfig meta-object configures the EDI data handler for the X.12 standard. Therefore, it provides an ISA attribute. If your EDI messages follow the EDIFACT standard, you must either add the UNA and UNB attributes to the default child meta object, or create a separate child meta-object for the EDIFACT standard, one that contains the UNA and UNB attributes instead of an ISA attribute.

The positional-information attribute specifies the information with the series of tags that Table 46 shows.

Table 46. EDI document information in the positional-information attribute

EDI document information	Attribute tag	Description	Required?
Segment separator	length	Specifies the length of the first segment (in number of characters), including the segment name but excluding the segment separator.	Yes
Segment count	seg_count	Specifies the relative position of the field that contains the number of segments written to the EDI document (during business-object-to-EDI conversion). For information about the use of this tag, see "Converting business objects to EDI documents" on page 102.	Yes
Composite separator	cs	Provides the relative position of the composite separator.	No
Repeat separator	rs	Provides the relative position of the repeat separator.	No
Transaction identifier	tid	Provides the relative position of the transaction ID.	Yes
DUNS number	duns	Provides the relative position of the DUNS number.	Yes
Version/release number	version	Provides the relative position of the functional group/message version number.	No

As Table 46 indicates, the positional-information attribute must provide values for the seg_count, length, tid, and duns tags and optionally the version tag. Specifying values for the cs and rs tags is not required. However, if either of these tags are omitted and the data handler parses EDI documents that contain composites, the data handler uses the precedence shown in Table 47 to obtain a value.

Table 47. Default values for composite and repeat separators

Precedence step		Composite separator	Repeat separator
1	Obtain the value of the corresponding meta-object attribute.	SEPARATOR_COMPOSIT	SEPARATOR_REPEAT
2	If the associated meta-object attribute is not set, use a hard-coded default.	colon (:)	caret (^)

The cs, rs, tid, and duns tags use the following format to indicate relative position within an EDI document:

tagname=seg_name+elem_pos+compos_pos

where:

- seg_name is the name of the segment in which the information is located.
- elem_pos is the position of the element within the *seg_name* segment. The numbering for element positions starts with "1" (not zero).
- compos_pos is optional; it specifies the position of the element within the *elem_pos* composite (assuming that *elem_pos* refers to a composite). The numbering for composite positions starts with "1" (not zero).

The seg_count tag uses the following format to indicate its relative position within an EDI document:

seg_count=*seg_name+elem_pos*

where seg_name and *elem_pos* are as described above; that is, the seg_count specification *never* includes a *compos_pos* value.

Note: Do not include spaces between the seg_name, *elem_pos*, and *compos_pos* values and the plus signs.

Figure 31 lists a sample EDI document that uses the X.12 standard. To improve readability of this EDI document, the example inserts newline characters at the end of each segment.

```
ISA*00*0000000000*02*XXXX*cw*ldtp3*cw*ld*970106*1525*U*00200*0000000100*0*P*<
GS*AA*ldtp3*ld*20010424*1525*142*X*004010
ST*846*001420001
SE*2*001420001
GE*1*142
IEA*1*0000000100
```

Figure 31. Sample EDI document in X.12 standard

To obtain the positional information from an EDI document that follows the X.12 standard, the EDI data handler takes the following steps:

1. Locate an attribute in the data handler's child meta-object that matches the name of the first segment.

   For the sample EDI document in Figure 31, the child meta-object needs to contain an ISA attribute (because the name of the first segment is ISA).
2. From this meta-object attribute, obtain the positional information.

   In the current example, the ISA attribute contains the following positional information:

   ```
   length=77;tid=ST+1;duns=ISA+6;seg_count=SE+1
   ```

   or (If version is included in the dbfile.txt) then

   ```
   length=77;tid=ST+1;version=GS+8;duns=ISA+6;seg_count=SE+1
   ```
3. Continue parsing the first segment of the document to determine the segment separator. The data handler assumes that the segment separator is at the end of the first segment so it takes the following steps:
   - Move to the end of this segment, based on the length of the first segment, provided by the length tag in the positional-information attribute of the child meta-object.

- Obtain the character at this position as the segment separator.

Note: Due to the algorithm that the data handler uses to locate the segment separator, this separator cannot be set to an alphanumeric character. In 2, the length tag specifies a segment length of 77 characters, which indicates that the segment separator in the Figure 31 document is the newline (carriage return) character. Therefore, the data handler interprets each newline character as a segment separator.

4. Continue parsing the first segment to determine the transaction ID, based on the `tid=` tag in the positional-information attribute of the child meta-object.

   The Figure 31 document follows the X.12 standard. This example does *not* include any composites. Therefore, its `ISA` attribute (in 2) does not include the composite (`cs`) or repeat (`rs`) separators. This attribute does include the `tid` tag, which specifies that the transaction ID occurs as the first element in the segment named `ST`; therefore, the transaction ID for the Figure 31 document is `846`.

5. Continue parsing the first segment to find the `version` (if the optional `version` tag is specified in the `dbfile.txt`).

   In Figure 31 the `version` is the 8th element of the `GS` segment: `004010`.

6. Parse the document to find the DUNS number. If the data handler cannot find the DUNS number, it logs an error and generates an exception.

   In 2, the `duns` tag specifies that the DUNS number occurs as the sixth element in the segment named `ISA`; therefore, the DUNS number for the Figure 31 document is `ldtp3`.

To obtain the positional information from an EDI document that follows the EDIFACT standard, the data handler takes the same steps as described for parsing an EDI document that follows the X.12 standard. The only major differences are:

- The data handler must locate a `UNB` attribute in the data handler's child meta-object (because the name of the first segment is `UNB`, not `ISA`).
- The data handler is more likely to find composites in parse an EDIFACT EDI document (composites rarely occur in X.12 documents). When composites occur, the data handler must parse the document for composite and repeat separators:
  - If the EDI document contains composites, the positional-information attribute of the child meta-object must contain at least the `cs` tag. It can also contain the `rs` tag, if the document uses a non-default repeat separator. The data handler determines the composite and repeat separator based on the positional information provided by the `cs` and `rs` tags. If the data handler cannot determine the separators, it uses one of the default values listed in Table 47.
  - If the EDI document does *not* contain composites, the positional-information attribute does not need to contain the `cs` or `rs` tag.

The following line is a fragment of an EDI document that follows the EDIFACT standard:

```
ST*st_child_value_1*,*st_grand_child_val_11,st_grand_child_val_12^
st_grand_child_val_13,st_grand_child_val_14*st_child_value_4*
st_grand_child_val_21,st_grand_child_val_22
```

Figure 32. Sample EDI document fragment with composite separator

If its first segment was named "UNB", then the child meta-object contains a UNB attribute that includes the following cs tag:

```
cs=ST+2;
```

This cs tag specifies that the composite separator occurs as the second element in the segment named ST; therefore, the data handler interprets the comma (,) as the composite separator. Suppose that the EDI document in which this fragment occurs did *not* specify a repeat separator; it uses the default value of the caret (^). Therefore, the UNB attribute in the child meta-object that this document uses does *not* need to contain an rs tag to specify the repeat separator. With no rs tag, the data handler assumes that the repeat separator has its default value. When the data handler encounters a caret (^), it interprets this character as the repeat separator.

To define a non-default repeat separator, the EDI document must include the non-default character in a field (usually in the header) and the positional-information attribute in the associated child meta-object must include the rs tag to indicate the location of this field.

Determining the name of the business object

A data handler can receive serialized data in one of two ways:

- Receive the serialized data *and* an empty business object: the data handler populates this business object with the serialized data.
- Receive *only* the serialized data: the data handler must create the business object before it can populate it with the serialized data.

Note: If the data handler receives a business object, it skips to the steps described in "Populating the business object" on page 111.

If the data handler does *not* receive a business object, the data handler must determine the type of the business object to create. The data handler calls the name handler, which takes the following steps:

1. Open the EDI name-handler lookup file based on the name given in the NameHandlerFile attribute of the child meta-object. This name-handler lookup file must already exist. If this open fails, the name handler generates an exception. For more information, see "Creating the name-handler lookup file" on page 91.
2. Check if the name-handler lookup file has been modified since the last time it was read. If it has, read the contents into the in-memory name-handler lookup table again.
3. Based on the transaction ID and the DUNS number (which were determined in the initialization phase), look up the name of the top-level EDI business object that is associated with this EDI document in the name-handler lookup table.

If the lookup of the business object name fails, the data handler logs an error and generates an exception. If the lookup is successful, the data handler creates a business object of the specified type to contain the data.

Note: These steps describe the behavior of the default name handler that is delivered with the EDI data handler. For information on how to create a custom name handler, see "Customizing the EDI data handler" on page 112.

Populating the business object

Once the EDI separators have been determined and the top-level business object has been created, the data handler takes the following steps to populate it with the serialized data:

1. If the `DefaultVerb` meta-object attribute is set, the data handler sets the verb in the business object to the value that `DefaultVerb` specifies. The delivered value for `DefaultVerb` is `Create`. Otherwise, the data handler assumes that no verb needs to be set.
2. The data handler determines if there are any child meta-objects (those whose names are listed in the `cw_mo_` tag of the business object application-specific information). The data handler does *not* perform the processing to populate these attributes of the business object. For more information about the `cw_mo_` tag, see "Implementing conversion from a business object" on page 177.
3. The data handler loops through the remaining attributes in the top-level business object definition. Based on the cardinality of each attribute, the data handler determines what part of the EDI document the attribute represents. For more information, see "Determining the attribute associated with the EDI data."
4. Once the data handler identifies the attribute associated with the current EDI data, it can take the appropriate steps to write the EDI data to this attribute. The data handler parses the EDI data based on the separators (which were determined in the initialization phase). For more information, see "Parsing the EDI document" on page 112.

Once the data handler has populated all attributes of the top-level business object, it can perform an optional check to ensure that all the EDI data has been parsed.

Determining the attribute associated with the EDI data

The structure of the business objects that hold EDI is determined by the EDI document specification. (For information on how to create this business object structure, see "Creating business object definitions for EDI documents" on page 101.) The EDI data handler uses the cardinality of the attribute to determine this attribute represents the current EDI part of the EDI document. Based on this cardinality, the data handler takes the following actions:

- If the attribute represents a *single-cardinality array*, the data handler examines the application-specific information of the attribute to determine the part of the EDI document associated with this attribute, as follows:
  - Examine metadata in the application-specific information of the attribute to determine the type of business object to create.
  - Create the business object indicated by the application-specific information of the attribute, as Table 48 shows.

Table 48. Application-specific information and the associated EDI business objects

Application-specific information	Child business object
`type=header`	Header business object
`name=segment name`	Segment business object
`name=name of first segment in loop;type=loop`	Segment-loop business object
`type=trailer`	Trailer business object

  - Recursively process this new child business object by looping through its attributes in the order they occur in the business object definition.
- If the attribute represents a *multiple cardinality array*, it represents a segment loop. The data handler recursively processes each child business object as if it

were a single-cardinality array. It creates a new business object in the array for each instance of the loop that occurs in the EDI document.

- If the attribute is of type `String`, the data handler generates an exception because the EDI structure dictates that all attributes of the top-level business object are either single-cardinality or multiple cardinality arrays.

Parsing the EDI document

The EDI data handler parses information in the EDI document based on the separators it has identified in the initialization phase. These separators determine each of the different pieces of data, which the data handler then matches to the appropriate attribute. Table 49 shows the parsing tasks that the data handler takes for the different EDI business objects.

Table 49. Parsing tasks for EDI business objects

Application-specific information	Parsing task
`type=header, type=trailer`	The data handler finds the position in the business object that corresponds to the next segment that appears in the document, and parses that segment to populate the child business object.
`name=`*`segment_name`* (no `type` tag)	The data handler assumes that the business object represents a segment and parses the current segment to populate the business object.
`type=loop`	The name of the first segment contained in the loop should be specified in the application-specific information. The data handler parses the EDI document for these loop segments and adds the data to the business object.

Customizing the EDI data handler

You can customize the EDI data handler by creating a special name handler. The EDI data handler calls the name handler to obtain the name of the business object to create. The data handler determines which name handler to invoke by using the value of the `NameHandlerClass` attribute stored in the data-handler meta-object. The default name handler included with the EDI data handler looks for the business object name in the name-handler lookup file (indicated by the `NameHandlerFile` meta-object attribute). If you need the name handler to function in a different way, you:

1. Create a custom name handler by extending the `NameHandler` class.
2. Configure the EDI data handler to use the custom name-handler class by updating the default value of the `NameHandlerClass` attribute in the meta-object for the EDI data handler.

For information on how to create a custom data handler, see “Building a custom name handler” on page 185.

Chapter 5. Request-Response data handler

The Request-Response data handler handles scenarios that require different data formats for their request and response business objects. The Request-Response data handler allows these two formats to be different. This chapter describes how the Request-Response data handler processes information and how to define business object definitions to be processed by the Request-Response data handler. It also discusses how to configure the Request-Response data handler.

This chapter contains the following sections:

Note: The Request-Response data handler is one of the base data handlers contained in the `CwDataHandler.jar` file. For information on how to install this data handler and where to store its source code, see Chapter 2, "Installing and configuring data handlers," on page 21.

Overview

The Request-Response data handler is a data-conversion module whose primary role is to provide support for request and response data that is in different formats; that is, it enables a calling context (such as an adapter or the Server Access Interface) to call two different data handlers:

- A *request data handler* handles data conversion for the WebSphere business integration system component that *initiates* a request.
- A *response data handler* handles data conversion for the WebSphere business integration system component that *responds* to a request.

With other data handlers, the calling contexts assumes that the data has the same format when sent to and received from the application or access client; therefore, the calling context is configured to call a single data handler to handle the conversion of both request and response business objects.

However, you might have a legacy application that accepts XML as an input and returns some custom-formatted document as output. Existing data handlers cannot easily handle this situation. However, you can configure the adapter that communicates with this legacy application to call the Request-Response data handler. This data handler can be configured to call separate data handlers for input and output:

- The IBM WebSphere Business Integration Data Handler for XML (XML data handler) to handle request business objects
- A custom data handler to handle response business objects

In request processing when the integration broker sends a request to this adapter, the adapter calls the Request-Response data handler, sending it the request business object. The Request-Response data handler checks its configuration to determine that it needs to invoke the XML data handler to convert this business object to an XML document. Once this business object has been converted, the Request-Response data handler returns XML document to the adapter, which in turn routes it to the legacy application.

Figure 33 shows an example of the business-object-to-string conversion as performed by the Request-Response data handler.

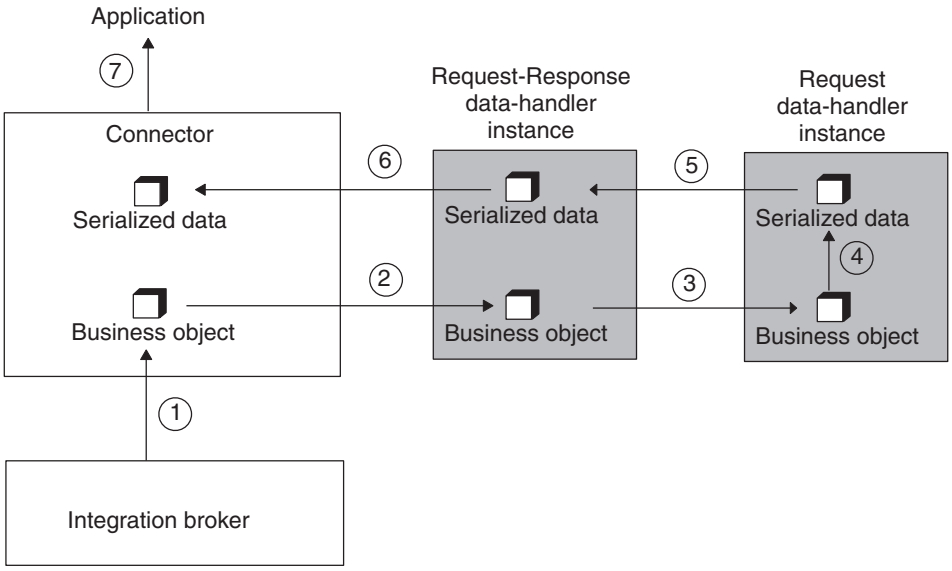

Figure 33. Business-object-to-string conversion with the Request-Response data handler

The adapter might later receive a response from the legacy application. This response would be in the legacy application's custom format. The adapter again calls the Request-Response data handler, sending it the response data. The Request-Response data handler checks its configuration to determine that it needs to invoke the custom data handler to convert this response data into a business object. Once this data has been converted, the Request-Response data handler returns business object to the adapter, which in turn routes it to the integration broker.

The Request-Response data handler is also to enable the ICS integration broker to post one business object type to a collaboration port and receive one or more different business objects in return. For example, an access client can send a customer object to a collaboration and receive an array of pending order objects for that customer in return.

The Request-Response data handler supports serialized data with the `text/requestresponse` MIME type. That serialized data may be either text or binary data. However, the default top-level meta-objects (`MO_DataHandler_Default` or `MO_Server_DataHandler`) do *not* support the `text/requestresponse` MIME type. Therefore, for an access client or connector to be able to call the Request-Response data handler, you must modify the appropriate top-level meta-object to support the `text/requestresponse` MIME type. For more information, see "Configuring the top-level meta-object" on page 123.

This overview provides the following information about the Request-Response data handler:

- "Request-Response data-handler components"
- "Features of the Request-Response data handler"
- "Processing request and response business objects" on page 119

Request-Response data-handler components

A data handler can receive serialized data in one of two ways:

- Receive the serialized data *and* an empty business object: the data handler populates this business object with the serialized data.
- Receive *only* the serialized data: the data handler must create the business object before it can populate it with the serialized data.

The Request-Response data handler uses a name handler to create the name of the top-level business object it creates. Figure 34 illustrates the Request-Response data handler components and their relationship to one another.

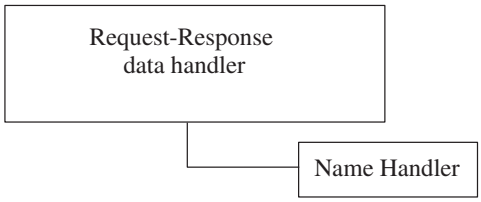

Figure 34. Request-Response data-handler components

The data handler invokes an instance of the name handler based on the value of the `NameHandlerClass` attribute in the Request-Response data handler child meta-object:

- If no class name is provided (the `NameHandlerClass` attribute is empty), the data handler uses its default name handler, which prepends the value of `BOPrefix` attribute to the name of the top-level business object generated by the data handler that is configured for requests. an error and generates an exception.

  For example, if the user specifies the default `BOPrefix` of `REQUESTTEST` and the request data handler generates the business object `Customer`, the Request-Response data handler creates a top-level object named `REQUESTTEST_Customer` and attempts to populate one of its child objects with the Customer object.
- If a class name is provided in the `NameHandlerClass` attribute, the Request-Response data handler uses this name handler to determine the business object name.

  To specify a custom name handler, set the `NameHandlerClass` to the name of the custom name-handler class. For information on how to create a custom name handler, see "Customizing the Request-Response data handler" on page 128.

The `NameHandlerClass` attribute in the version of the meta-object delivered with the product is blank. Therefore, the Request-Response name handler uses its default name handler.

Features of the Request-Response data handler

The Request-Response data handler is useful in both of the following cases:

- "Support for event processing" on page 116

• "Support for request processing" on page 118

Support for event processing

Event processing involves the notification of the integration broker that some event, which indicates a change to application business entities, has occurred. In event notification, the calling context of a data handler calls a data handler: the data handler to convert the serialized data to a business object (which is then routed to the integration broker). This string-to-business-object conversion is performed by the request data handler, because this data handler handles conversion from the request (input) format to a business object.

Event processing can be either synchronous or asynchronous. However, because in asynchronous event processing the adapter ((in particular, the connector component of the adapter) does not wait for a response from the integration broker , as follows:

- Synchronous event processing occurs when an access client notifies the IBM WebSphere InterChange Server (in particular, some collaboration within this server) that some event has occurred. The access client does not wait for a response from ICS, nor does it wait for a response. In this case, the Server Access Interface (within ICS) invokes the necessary data handlers.
- Asynchronous event processing occurs when an adapter (in particular, the connector component of the adapter) receives an event from its application or technology and sends this event (in the form of a business object) to an integration broker to notify it that something has occurred. However, the connector does not wait for a response from the integration broker. Therefore, the Request-Response data handler is not useful in the case of asynchronous event processing.

Note: For more information on the calling contexts of data handlers, see "Contexts for calling data handlers" on page 6.

For example, the following steps describe synchronous event processing. They describe how the Request-Response data handler allows the Server Access Interface (within ICS) to send data in one format (its request format) to the ICS and to receive a different form of data (its response format) in return. This allows an access client to complete a scenario such as sending a customer XML document and receiving an XML document containing pending orders for that customer in return.

1. An access client sends an event (as data in the request format) to be executed by a collaboration within ICS.
2. ICS creates a new instance of the Request-Response data handler and passes it request-format data.
3. The Request-Response data handler calls its configured request data handler, which converts the request-format data to a business object.

   The Request-Response data handler uses its `getBO()` method to handle the serialized data it receives. It invokes the request data handler on this serialized data if either of the following conditions are true:

   - If the `getBO()` method receives *only* the serialized data as an argument, it considers this data as a new request.
   - If the `getBO()` method receives top-level business object *and* the serialized data as an argument, it checks the child business objects of the top-level business object to determine its next action.

If all child business objects contain a value of CxIgnore (indicating that they are *empty*), the data handler assumes that the serialized data represents a new request.

Note: If a child business object within the top-level business object is populated, the data handler assumes that this child object represents the original request and consequently that the serialized data represents the response. Therefore, it creates an instance of the response data handler to process the data. For more information, see "Support for request processing" on page 118.

In either of these cases, the Request-Response data handler creates an instance of the request data handler and uses this data handler to convert the request-format data to a request business object. To the request data handler, the Request-Response data handler passes the request-format data. The request data handler returns the corresponding request business object.

4. The Request-Response data handler adds the request business object as a child of the top-level business object, as follows:
   - If the data handler's `getBO()` method did *not* receive a top-level business object, it must creates a new one and then add the request business object to it. To name this new business object, the data handler calls its name handler. For more information, see "Request-Response data-handler components" on page 115.
   - If the data handler's `getBO()` method *did* receive a top-level business object, it just adds the request business object to it.
5. The Request-Response data handler returns the top-level business object synchronously to the collaboration (which the access client has specified).
6. The collaboration receives the top-level business object, extracts the child object (which contains the request business object) and executes some business process.
7. The collaboration creates a new response business object and adds it to the top-level business object, then returns successfully.

   This top-level business object is the one that the collaboration has received from the Request-Response data handler. Once the collaboration has updated this business object, the business object contains both the original request business object and the newly created response business object.
8. ICS passes the modified top-level business object to the Request-Response data handler.
9. The Request-Response data handler calls its configured response data handler to convert the response business object to response-format data.

   The Request-Response data handler uses its `getStringFromBO()` method to handle the top-level business object it receives. If more than one child business object in the top-level business object is populated, the data handler assumes that the top-level business object contains *both* the original request business object and its response business object and consequently that the response business object must be converted. Therefore, it creates an instance of the response data handler to process the last defined child business object within the top-level business object as the response business object.

   Note: If only one child business object within the top-level business object that `getStringFromBO()` received is populated, the data handler assumes that the child object represents a new request. Therefore, it creates an instance of the request data handler to convert the request business

object to serialized data (in the request format). For more information, see "Support for request processing."

10. The Request-Response data handler returns the response-format data to its caller (Server Access Framework in ICS).
11. ICS returns the response-format data (which is based on the response business object) to the access client.

A connector can also perform synchronous event processing with the `executeCollaboration()` method. Generally, however, it performs asynchronous event processing using an event detection mechanism such as polling.

Support for request processing

Request processing involves the receipt of requests from the integration broker and to initiate the appropriate changes in the application business entities. Unlike event processing, which can be initiated by either an access client (synchronously) or a connector (asynchronously), request processing is initiated by an integration broker and involves communication with connectors only (not access clients).

In request processing, the calling context of a data handler calls a data handler to convert the serialized data to a business object (which is then routed to the integration broker). This string-to-business-object conversion

For example, the following steps describe request processing that involves a IBM WebSphere InterChange Server integration broker and a technology adapter. This technology adapter is configured to use the Request-Response data handler to handle request and response data. In this case, the request data is in one format and the response data in another.

1. The collaboration creates a new instance of the top-level business object and adds the request business object as a child.
2. The collaboration sends a request (in the form of the top-level business object) to the technology connector, which in turn passes the top-level business object to the Request-Response data handler.
3. The Request-Response data handler calls its configured request data handler to convert the request business object to request-format data.

   The Request-Response data handler uses its `getStringFromBO()` method to handle the top-level business object it receives. If only one child business object within the top-level business object is populated, the Request-Response data handler assumes that the child object represents a new request. Therefore, it creates an instance of the request data handler to convert the request business object to serialized data (in the request format).

   Note: If more than one child business object in the top-level business object that `getStringFromBO()` received is populated, the data handler assumes that the top-level business object contains *both* the original request business object and its response business object and consequently that the response business object must be converted. Therefore, it creates an instance of the response data handler to process the last defined child business object within the top-level business object as the response business object. For more information, see "Support for event processing" on page 116.
4. The technology connector sends the request-format data to the application.
5. The application performs some tasks and returns response-format data to the technology connector.

6. The technology connector passes both the original top-level business object and the response-format data to the Request-Response data handler.
7. The Request-Response data handler calls its configured response data handler to convert the response-format data to a business object.

   The Request-Response data handler uses its `getBO()` method to handle the top-level business object and the serialized data it receives. If any of the child business objects within the top-level business object are populated, the Request-Response data handler assumes that the child object represents the original request and consequently that the serialized data it received is in the response format. Therefore, it creates an instance of the response data handler to convert the serialized data to a response business object.

   Note: If all child business objects in the top-level business object that `getBO()` receives contain a value of CxIgnore (indicating that they are *empty*), the data handler assumes that the serialized data represents is a new request and creates an instance of the request data handler to process the data. For more information, see "Support for event processing" on page 116.
8. The Request-Response data handler adds this response business object as a child of the top-level business object and then returns this top-level business object to its caller (the technology connector).
9. The technology connector returns the updated top-level business object to the collaboration.
10. The collaboration receives the top-level business object and incorporates the content of its response business object into the business process.

Processing request and response business objects

Use of the Request-Response data handler to convert a request business object to its appropriate request format, or to convert data in a response format to a response business object requires that the steps outlined in Table 50 occur.

Table 50. Using the Request-Response data handler

Step	For more information
1. Business object definitions that describe the business-object structure must exist and be available to the Request-Response data handler (and its component data handlers) when it executes.	"Requirements for business object definitions" on page 120
2. The Request-Response data handler must be configured for your environment.	"Configuring the Request-Response data handler" on page 123
3. The Request-Response data handler must be called from a calling context (connector or access client) to perform the appropriate data operation:	
a) Data operation: Receive the request from the component that initiates this request and convert it to the appropriate format.	"Converting business objects with the request data handler" on page 126
b) Data operation: Receive the response from the component that responds to the request and convert it to the appropriate format.	"Converting business objects with the response data handler" on page 127

Requirements for business object definitions

To use the Request-Response data handler, you must create or modify business object definitions so that they provide the structure that the data handler requires. However, unlike other data handlers, you do not need to modify business object definitions so that they contain metadata. This section provides the information you need to create business object definitions to work with the Request-Response data handler. In particular, it provides the following information:

- "Understanding Request-Response business object structure"
- "Creating business object definitions for the Request-Response data handler" on page 122

Understanding Request-Response business object structure

The Request-Response data handler uses business object definitions when it receives business object from or sends business objects to its request or response data handler. The Request-Response data handler places specific requirements on the structure of business objects it can process. Business objects passed to the data handler must contain one request child object and one or more response child objects. These child objects must conform to the requirements of the data handler that will process them.

Note: The Request-Response data handler does *not* require application-specific information in the business object definitions or their attributes.

Figure 35 shows the structure of the business objects that represent a Request-Response business object.

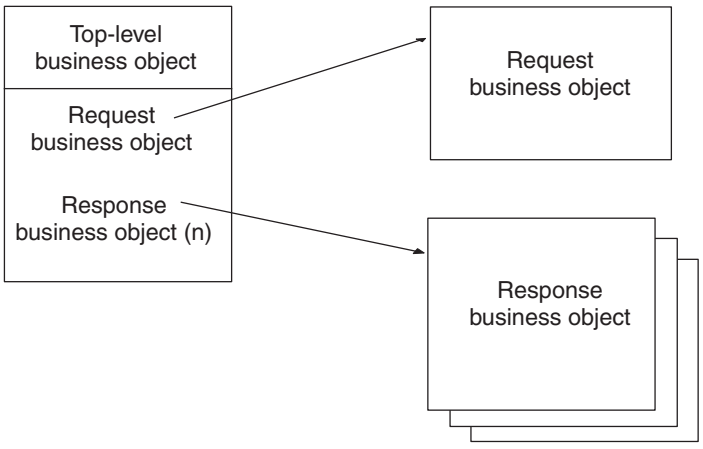

Figure 35. Business object structure for a request-response business object

To ensure that business object definitions conform to the requirements of the Request-Response data handler, use the guidelines provided for each of the following business objects:

- "Top-level business object" on page 121
- "Request business object" on page 121
- "Response business object" on page 122

Top-level business object

The Request-Response data handler expects a top-level business object to hold the information it receives from or sends to its calling context. Table 51 describes how the Request-Response data handler interprets the properties of a business object and describes how to set the properties when modifying a business object for use with the Request-Response data handler.

Table 51. Properties for the top-level business object definition

Property name	Description
Name	Each business object definition must have a unique name. It is recommended that these business object definitions begin with a standard prefix. The name of the top-level business object depends on the message standard, as follows: *BusObj Prefix + Response Business Object*
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific Information	No application-specific information is required.

This top-level business object must contain the following attributes:

- A single-cardinality attribute to hold a single request business object

  This request business object must conform to any business-object requirements that the request data handler requires.
- An single-cardinality attribute to hold the response business objects

  The response business object must conform to any business-object requirements that the response data handler requires.

  Notes:

  1. If the target application could return *more than one* kind of response business object, the top-level business object must contain one child business object for each type of response business object.

     For example, if the target application could return either a Customer XML document or an OrderUpdate XML document, the top-level business object definition must include two attributes, one to hold the business object definition that represents the Customer XML document, and the second to hold the businesss object definition that holds the OrderUpdate XML document. When it received the response business object, the data handler would populate the appropriate attribute.
  2. The data handler fails if it receives a top-level business object that does *not* conform to expectations or if other data handlers involved fail to convert the business objects or documents passed to them.

Request business object

To hold the request information for the request data handler, the Request-Response data handler expects a request business object as the first attribute of the top-level business object. This attribute should be of single cardinality. Table 52 describes how the Request-Response data handler interprets the properties of this business object definition and describes how to set these properties when modifying the business object for use with the Request-Response data handler.

Table 52. Properties for the request business object definition

Property name	Description
Name	Each business object definition must have a unique name. This name must match the name of the business object definition that the request data handler handles.
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific information	Depends on the particular request data handler being used.

Note: For information on the format of the request business object, see the documentation for the data handler that acts as the request data handler.

For example, if you specified `RequestDataHandlerMimeType` as `text/xml`, the child object you define as your request business object must be compatible with the XML data handler.

Response business object

To hold the response information for the response data handler, the Request-Response data handler expects a response business object in the attributes starting with the second attribute of the top-level business object. This attribute should be of single cardinality. If the response data handler might return more than one type of business object, the top-level business object has an attribute for each possible type. Table 53 describes how the Request-Response data handler interprets the properties of this business object definition and describes how to set these properties when modifying the business object for use with the Request-Response data handler.

Table 53. Properties for the response business object definition

Property name	Description
Name	Each business object definition must have a unique name. This name must match the name of the business object definition that the response data handler handles.
Version	A constant representing the current version of the business object definition. Current value is 1.0.0.
Application-specific information	Depends on the particular request data handler being used.

Note: For information on the format of the request business object, see the documentation for the data handler that acts as the request data handler.

For example, if you specified `ResponseDataHandlerMimeType` as `text/abc`, the child object you define as your request business object must be compatible with a custom data handler that can handle the `abc` MIME type.

Creating business object definitions for the Request-Response data handler

This section describes how to create business object definitions to represent the structure that the Request-Response data handler expects. Use Business Object Designer to add or delete attributes from the business object definition as well as edit attribute properties, as needed.

As described in "Understanding Request-Response business object structure" on page 120, the Request-Response data handler requires that you create the following business object definitions:

- "Creating the top-level business object definition"
- "Creating other business object definitions"

Creating the top-level business object definition

To create the top-level business object definition for the Request-Response data handler, you must manually create a business object definition using Business Object Designer:

1. Create the top-level business object definition.

   For information on the structure of this top-level business object, see "Top-level business object" on page 121.
2. Create the child business objects for the top-level business object. In the top-level business object, create a child object attribute for the business objects shown in Table 54.

Table 54. Business objects for the Request-Response data handler

Attribute	Notes	Business object
Request business object	Contains the information about the request.	"Request business object" on page 121
Response business object	Contains the information about the request's response	"Response business object" on page 122

Creating other business object definitions

To create the request and response business object definitions, you can use either of the following ways:

- You can use an Object Discovery Agent (ODA) to export the serialized data as a business object definition. ODAs exist for several data formats, including XML documents.

  For information on the XML ODA, see "Using the XML ODA," on page 209. For information on other ODAs, refer to the corresponding adapter guide.
- You can manually create a business object definition for the data using Business Object Designer.

  For more information, see "Creating the top-level business object definition"

Configuring the Request-Response data handler

The Request-Response data handler retrieves its configuration properties from a hierarchy of meta-objects, as follows:

- The parent meta-object allows a connector or Server Access Interface to instantiate a data handler based on the MIME-type of the document.
- The child meta-object contains all the information needed to process the data in the document, including the class name of the data handler, the prefix of the business object to create, and more.

Configuring the top-level meta-object

The MIME type contained in the parent meta-object indicates which MIME types are supported and which data handlers provide this support. Neither of the provided top-level meta-object include an entry for the Request-Response data handler. To enable your connector or an access client to use the Request-Response data handler, you must add an attribute for the `text/requestresponse` MIME type

to the MO_Server_DataHandler or MO_DataHandler_Default top-level meta-object. This attribute must be of type MO_DataHandler_DefaultRequestResponseConfig.

The following fragment of a business object definition shows the definition for the text/requestresponse attribute:

```
[Attribute]
Name = text.requestresponse
Type = MO_DataHandler_DefaultRequestResponseConfig
ContainedObjectVersion = 1.0.0
Relationship = Containment
Cardinality = 1
MaxLength = 1
IsKey = false
IsForeignKey = false
IsRequired = false
IsRequiredServerBound = false
[End]
```

Note: For more information on top-level meta-objects and how to modify them, see "Top-level meta-objects" on page 24.

Configuring the child meta-object

To configure a Request-Response data handler, you must ensure that its configuration information is provided in the Request-Response's child meta-object.

Note: To configure a Request-Response data handler, you must also create or modify business object definitions so that they support the data handler. For more information, see "Requirements for business object definitions" on page 120.

For the Request-Response data handler, IBM delivers the default child meta-object MO_DataHandler_DefaultRequestResponseConfig. Each attribute in this meta-object defines a configuration property for the Request-Response data handler. Table 55 describes the attributes in this child meta-object.

Table 55. Child meta-object attributes for the Request-Response data handler

Attribute name	Description	Delivered default value
BOPrefix	Prefix used by the default NameHandler class to build the name of the top-level business object. The default value must be changed to match the name of the associated the business object definition. The attribute value is case-sensitive.	REQUESTTEST
ClassName	Name of the data-handler class to load for use with the specified MIME type. The top-level data-handler meta-object must has an attribute whose name matches the specified MIME type and whose type is the Request-Response child meta-object (described by this table).	com.crossworlds. DataHandlers.text. requestresponse
NameHandlerClass	Name of the name-handler class to use to determine the name of the top-level business object from the content of a request document. Change the default value of this attribute if you create your own custom name handler. For more information, see "Building a custom XML name handler" on page 86.	com. crossworlds. DataHandlers.xml. TopElementNameHandler

Table 55. Child meta-object attributes for the Request-Response data handler (continued)

Attribute name	Description	Delivered default value
RequestDataHandlerMimeType	The MIME type of requests processed by this data handler. The Request-Response data handler uses this MIME type to determine the data handler to instantiate for processing any request business objects or documents.	text/xml
ResponseDataHandlerMimeType	The MIME type of responses processed by this data handler. The Request-Response data handler uses this MIME type to determine the data handler to instantiate for processing any response business objects or documents.	text/xml
ObjectEventId	Placeholder not used by the data handler but required by the business integration system.	*None*

The "Delivered default value" column in Table 55 lists the value in the Default Value property for the corresponding attribute in the delivered business object. You must examine your environment and set the Default Value properties of those attributes to the appropriate values. You must make sure that at least the ClassName and BOPrefix attributes have default values.

To create the MO_DataHandler_DefaultRequestResponseConfig child meta-object, use Business Object Designer to create a business object definition with the following format:

```
[BusinessObjectDefinition]
Name = MO_DataHandler_DefaultRequestResponseConfig
Version = 1.0.0
   [Attribute]
   Name = ClassName
   Type = String
   Cardinality = 1
   MaxLength = 255
   IsKey = false
   IsForeignKey = false
   IsRequired = false
   DefaultValue = com.crossworlds.DataHandlers.text.requestresponse
   IsRequiredServerBound = false
   [End]
   [Attribute]
   Name = NameHandlerClass
   Type = String
   MaxLength = 255
   IsKey = false
   IsForeignKey = false
   IsRequired = false
   IsRequiredServerBound = false
   [End]
   [Attribute]
   Name = RequestDataHandlerMimeType
   Type = String
   MaxLength = 255
   IsKey = false
   IsForeignKey = false
   IsRequired = false
   DefaultValue = text/xml
   IsRequiredServerBound = false
   [End]
```

```
    [Attribute]
    Name = ResponseDataHandlerMimeType
    Type = String
    MaxLength = 255
    IsKey = false
    IsForeignKey = false
    IsRequired = false
    DefaultValue = text/xml
    IsRequiredServerBound = false
    [End]
    [Attribute]
    Name = BOPrefix
    Type = String
    MaxLength = 255
    IsKey = false
    IsForeignKey = false
    IsRequired = false
    DefaultValue = Wrapper
    IsRequiredServerBound = false
    [End]
    [Attribute]
    Name = DummyKey
    Type = String
    MaxLength = 1
    IsKey = true
    IsForeignKey = false
    IsRequired = false
    DefaultValue = 1
    IsRequiredServerBound = false
    [End]
    [Attribute]
    Name = ObjectEventId
    Type = String
    Cardinality = 1
    MaxLength = 255
    IsKey = false
    IsForeignKey = false
    IsRequired = false
    IsRequiredServerBound = false
    [End]
    [Verb]
    Name = Create
    [End]
    [Verb]
    Name = Delete
    [End]
    [Verb]
    Name = Retrieve
    [End]
    [Verb]
    Name = Update
    [End]
[End]
```

Refer to “Configuring data handlers” on page 24 for information on where to put this child meta-object file.

Converting business objects with the request data handler

The *request data handler* handles data conversion for the WebSphere business integration system component that initiates a request:

- In request processing, the integration broker initiates the request, in the form of a request business object sent to a connector.

  WebSphere InterChange Server
  When the integration broker is InterChange Server (ICS), the collaboration creates the request business object and sends it to the appropriate connector for request processing.

  Therefore, the request data handler performs a business-object-to-string conversion on the request, converting the request business object to serialized data. This serialized data is in the *request format*, which is the format that the connector's application (or access client) accepts as input.
- In event processing, the calling context (access client or connector) initiates the request, in the form of serialized data sent to the integration broker.

  WebSphere InterChange Server
  When the integration broker is ICS, you can use an access client to initiate synchronous event processing. For more information, see "Support for event processing" on page 116.

  Therefore, the request data handler performs a string-to-business-object conversion on the request, converting the serialized data to a request business object. This serialized data is in the request format, which is the format that the access client or the connector's application generates as output.

The Request-Response data handler determines which data handler to invoke as its request data handler based on the `RequestDataHandlerMimeType` property in the child meta-object. The `RequestDataHandlerMimeType` property contains the MIME type that the request data handler supports. If `RequestDataHandlerMimeType` has *not* been initialized, the Request-Response data handler logs an error and generates an exception. Therefore, you *must* initiauses the `RequestDataHandlerMimeType` property.

Converting business objects with the response data handler

The *response data handler* handles data conversion for the WebSphere business integration system component that responds to a request:

- In request processing, the connector's application responds to the request, in the form of serialized data sent to connector, which in turn routes it to the integration broker.

  WebSphere InterChange Server
  When the integration broker is InterChange Server (ICS), the collaboration receives the response business object from the connector.

  Therefore, the response data handler performs a string-to-business-object conversion on the response, converting the serialized data to a response business object. This serialized data is in the *response format*, which is the format that the connector's application (or access client) generates as output.

- In event processing, the integration broker responds to the request, in the form of a response business object sent to the calling context (access client or connector).

  WebSphere InterChange Server
  When the integration broker is ICS, the collaboration within ICS generates the response business object. ICS then routes this response business object back to the calling context. For more information, see "Support for event processing" on page 116.

  Therefore, the response data handler performs a business-object-to-string conversion on the response, converting the response business object to serialized data. This serialized data is in the response format, which is the format that the access client or the connector's application accepts as input.

The Request-Response data handler determines which data handler to invoke as its response data handler based on the `ResponseDataHandlerMimeType` property in the child meta-object. The `ResponseDataHandlerMimeType` property contains the MIME type that the response data handler supports. If `ResponseDataHandlerMimeType` has *not* been initialized, the Request-Response data handler logs an error and generates an exception. Therefore, you *must* initiauses the `ResponseDataHandlerMimeType` property.

Error handling

The Request-Response data handler throws an exception if it cannot process a request. The request-response data handler provides an exception message that describes both what the data handler was attempting to do when the error occurred and any message returned by a component involved in the transaction (such as another data handler or the JCDK). These exceptions are propagated (perhaps not as an exception but another data structure) to the component that invoked the data handler initially.

The Request-Response data handler logs error and traces messages using the services provided by the data handler framework. Error messages and warnings are recorded in ICS logs. The data-handler does not log any messages to the Java console.

Customizing the Request-Response data handler

You can customize the Request-Response data handler by creating a special name handler. The Request-Response data handler calls the name handler to obtain the name of the business object to create. The data handler determines which name handler to invoke by using the value of the `NameHandlerClass` attribute stored in the data-handler meta-object. The default name handler included with the Request-Response data handler prepends the value of the `BOPrefix` attribute to the name of the business object that the response data handler returns. If you need the name handler to function in a different way, you:

1. Create a custom name handler by extending the `NameHandler` class.
2. Configure the Request-Response data handler to use the custom name-handler class by updating the default value of the `NameHandlerClass` attribute in the meta-object for the Request-Response data handler.

For information on how to create a custom data handler, see “Building a custom name handler” on page 185.

Chapter 6. FixedWidth data handler

The FixedWidth data handler converts business objects to fixed-width strings and streams and from fixed-width strings and streams to business objects. This chapter describes how the FixedWidth data handler processes fixed-width documents and how to define business objects to be processed by the data handler. You can use this information as a guide to implementing business objects that conform to the requirements of the FixedWidth data handler. This chapter also discusses how to configure the FixedWidth data handler. This chapter contains the following sections:

Note: The FixedWidth data handler is one of the base data handlers contained in the `CwDataHandler.jar` file. For information on how to install this data handler and where to store its source code, see Chapter 2, "Installing and configuring data handlers," on page 21.

Overview

The FixedWidth data handler is a data-conversion module whose primary role is to convert business objects to and from strings or streams that have a format of fixed-width fields. A fixed-width string or stream is serialized data with the `text/fixedwidth` MIME type. The data handler parses text data using fixed-width fields. The field lengths are specified by the MaxLength property of each business object attribute. The value of this property is stored in the business object definition.

The default top-level connector meta-object (`MO_DataHandler_Default`) supports the `text/fixedwidth` MIME type. Therefore, a connector that is configured to support the `MO_DataHandler_Default` data-handler meta-object can call the FixedWidth data handler. For an access client to be able to call this data handler when using the InterChange Server integration broker, you must modify the `MO_Server_DataHandler` meta-object to support the `text/fixedwidth` MIME type. For more information, see "Modifying the top-level meta-object" on page 188.

Features of the FixedWidth data handler

The FixedWidth data handler uses the value of the MaxLength property of attributes in a business object definition to determine how to read and write data. `MaxLength` is defines the maximum number of characters of the attribute value, including padding to allow for right-justified or left-justified text.

You can configure a pad character that represents spaces to add to or remove from the data for alignment. Pad characters are added when converting business objects to strings and removed when converting strings to business objects. You can also configure the alignment of a business object attribute value to be left- or right-justified. This makes it possible for the attribute value data to retain meaningful characters. Table 56 describes the values for the `Alignment` meta-object attribute.

Table 56. Values of the Alignment meta-object attribute

Value	Description
LEFT	Trims left side and pads right side.
RIGHT	Trims right side and pads left side.
BOTH	When converting strings to business objects, trims both right and left sides. When converting business objects to strings, pads right side with pad characters.

In addition, with the `Truncation` meta-object attribute, you can configure the FixedWidth data handler to truncate business object values to their `MaxLength`, or generate an error if an attribute value is a string longer that `MaxLength`. You can also set the size of the length to be used for the business object name, verb, and business object count for cardinality 1 or n child objects.

Use Business Object Designer to set the value of the `MaxLength` property of String attributes. To change the value of the MaxLength property of other types (such as Integer, Double, and so forth), you must save the business object definition to a file, edit the file manually, and then import the modified definition into the business integration system.

Business object and FixedWidth string processing

The FixedWidth data handler performs the operations listed in Table 57.

Table 57. Data operations for the FixedWidth data handler

Data-handler operation	For more information
Receives a business object from the caller, converts the business object into a FixedWidth string or stream, and passes the FixedWidth data to the caller.	"Converting business objects to FixedWidth documents" on page 135
Receives a string or stream from the caller, builds a business object, and returns the business object to the caller.	"Converting FixedWidth documents to business objects" on page 136

Configuring the FixedWidth data handler

To configure the FixedWidth data handler, take the following steps:

- Enter the appropriate values for the attributes of the FixedWidth child meta-object.
- Create or modify business object definitions so that they support the data handler.

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

Configuring the FixedWidth child meta-object

To configure a FixedWidth data handler, you must ensure that its configuration information is provided in the FixedWidth child meta-object. For the FixedWidth data handler, IBM delivers the `MO_DataHandler_DefaultFixedWidthConfig` child meta-object. Each attribute in this meta-object defines a configuration property for the FixedWidth data handler. Table 58 describes the attributes for this child meta-object.

Table 58. Child meta-object attributes for the FixedWidth data handler

Meta-object attribute name	Meaning	Delivered default value
ClassName	Name of the data handler class to load for use with the MIME type that matches the name of the attribute in the top-level data-handler meta-object. This attribute has the FixedWidth child meta-object as its type.	com.crossworlds.DataHandlers.text.fixedwidth
Alignment	Adds or removes the PadCharacter attribute. For event processing, pad characters are trimmed. For request processing, pad characters are added. Possible values are BOTH, LEFT, and RIGHT. For example, "LEFT" alignment means the value of the business object attributes moves to the extreme left of the space for that attribute value. "BOTH" alignment for event notification means that pad characters are trimmed off both the left and right sides. "RIGHT" alignment for request processing means that the right side is padded with pad characters.	BOTH
BOCountSize	Specifies the space allocated for the total number of business objects being processed.	3
BONameSize	Specifies the space allocated for the name of the business object.	50
BOVerbSize	Specifies the space allocated for the verb.	20
CxBlank	When converting from a business object, the FixedWidth data handler writes the value configured for the Default Value property of the CxBlank meta-object attribute to the fixed-width document whenever it encounters a business object attribute whose value is CxBlank. When converting to a business object, the FixedWidth data handler assigns the value configured for the Default Value property of the CxBlank meta-object attribute to the business object attribute's value whenever it encounters the value of this CxBlank meta-object attribute in the fixed-width document. Business objects must have at least one primary key that does *not* contain the value CxBlank at runtime.	CxBlank value
CxIgnore	When converting from a business object, the FixedWidth data handler writes the value configured for the Default Value property of the CxIgnore meta-object attribute to the fixed-width document whenever it encounters a business object attribute whose value is CxIgnore. When converting to a business object, the FixedWidth data handler assigns the value configured for the Default Value property of the CxIgnore meta-object attribute to the business object attribute's value whenever it encounters the value of this CxIgnore meta-object attribute in the fixed-width document. Business objects must have at least one primary key that does *not* contain the value CxIgnore at runtime.	CxIgnore value
DummyKey	Key attribute required by the business integration system.	1
OmitObjectEventId	Boolean value to determine whether or not to include ObjectEventId data in business-object-to-string and string-to-business-object conversions.	false
PadCharacter	Indicates spaces to add or remove for alignment. You can specify any character as the pad character.	#
Truncation	Sets removal of characters. If true, any attribute value in the business object that is greater than MaxLength is truncated to MaxLength during request processing. If false, an error is logged and formatting stops.	false
ObjectEventId	Placeholder not used by the data handler but required by the business integration system.	none

The "Delivered default value" column in Table 58 lists the value in the Default Value property for the corresponding attribute in the delivered business object. You must examine your environment and set the Default Value properties of those attributes to the appropriate values for your system and your FixedWidth documents. You must make sure that at least the `ClassName` attribute has a default value.

Note: Use Business Object Designer to modify business object definitions.

Business objects requirements

The FixedWidth data handler makes assumptions about the structure of the business objects that it handles. Therefore, when you create a business object for conversion using the FixedWidth data handler, follow these rules:

- Make sure that every attribute in the business object definition has an appropriate `MaxLength` property value. This ensures that the FixedWidth data handler can properly process the conversion of data from a business object to a FixedWidth format and from a FixedWidth format to a business object.
- Make sure that the `ObjectEventId` attribute is included in every business object at all levels of a business object hierarchy. Business Object Designer does this automatically when it saves a business object definition, but you should confirm that the requirement is met.

Business object structure

There are no requirements regarding the structure of the business objects for the FixedWidth data handler. The data handler can process any business object as long as the `MaxLength` attribute property has a value.

The business objects that the data handler processes can have any name allowed by the business integration system.

Business object attribute properties

Business object architecture contains various properties that apply to attributes. Table 59 describes how the FixedWidth data handler interprets these properties and describes how to set the properties when modifying a business object.

Table 59. Attribute properties for business objects converted using the FixedWidth data handler

Property name	Description
Name	Each business object attribute must have a unique name.
Type	Each business object attribute must have a type, such as Integer, String, or the type of a contained child business object.
Key	Not used by the FixedWidth data handler.
MaxLength	Determines the width of the field in which the attribute value is included.
Foreign Key	Not used by the FixedWidth data handler.
Required	Not used by the FixedWidth data handler.
Default Value	Not used by the FixedWidth data handler.
Cardinality	Supports cardinality 1 and cardinality n objects.

Business object application-specific information

The FixedWidth data handler does not require any application-specific information in business objects or their attributes. The data handler does, however, check for the existence of the `cw_mo_` tag, which a business object might use to indicate any child meta-object that the connector uses. The data handler ignores any attribute

identified by the cw_mo_ tag in the application-specific information of the business object. For more information about the cw_mo_ tag, see "Implementing conversion from a business object" on page 177.

Using existing business object definitions

The FixedWidth data handler can convert any business object to a FixedWidth string as long as the business object delivers data in a form that complies with the requirements of the data handler. The single requirement of the FixedWidth data handler is that each business object attribute have a MaxLength value specified. Existing business objects may need to be modified to specify an appropriate value for MaxLength.

Although existing business objects that meet this requirement can be converted by the FixedWidth data handler, a good practice is to create your own business objects for each type of data to be processed. If you use a sample business object, or a business object developed to support the same application in another implementation, be sure to modify the definition as necessary to include only the attributes required for the implementation for which you are developing.

Therefore, to convert existing business objects to a form that closely corresponds to your data, modify the business object to provide only the data required by the application and the information required by the data handler. To adapt existing business objects for use with the FixedWidth data handler, do the following:

1. Perform a functional analysis of the target application, and compare the results to existing business objects to determine the required fields of a business object definition.
2. Use Business Object Designer to add or delete attributes from the business object definition as needed.

Converting business objects to FixedWidth documents

To convert a business object to a FixedWidth document, the FixedWidth data handler loops through the attributes of the business object in sequential order. It generates fields in a fixed-width string recursively in the order in which attributes appear in the business object and its children.

The FixedWidth data handler processes business objects into a FixedWidth document as follows:

1. The data handler creates a fixed-width string to contain the data in the business object.
2. The data handler adds the business object name and the verb to the fixed-width string. The name of the business object can be specified as an argument to the conversion method.
3. The data handler examines the application-specific information in the business object definition to determine if there are any child meta-objects (those whose names are listed in the cw_mo_ tag of the business object application-specific information). The data handler does not include these attributes in the FixedWidth document. For more information about the cw_mo_ tag, see "Implementing conversion from a business object" on page 177.
4. The data handler looks for the meta-object attribute named OmitObjectEventId. If this is set to true, the data handler does not include the ObjectEventId data of the business object in the FixedWidth document.

5. The data handler loops through the remaining business object attributes in order, adding the correct padding to the string for each simple attribute. For array attributes, the data handler does the following:
   - If the attribute represents a single-cardinality attribute, the data handler adds the attribute name and a count of 1 to the string, and then recursively processes the child business object to add the values of each attribute to the string.
   - If the attribute represents a multiple cardinality array, the data handler adds the attribute name and the child object count to the string, and then recursively processes each child business object, adding the values of each attribute to the string.
6. When the data handler completes the conversion, it returns the serialized data to the caller. The data handler returns the data in the form (`String` or `InputStream`) requested by the caller.

Note: Any attribute value in the business object that has a length greater than `MaxLength` is truncated to `MaxLength` during request processing if the value of the Default Value property of the `Truncation` meta-object attribute is set to `true`. If `Truncation` is set to `false` and an attribute value has a length greater than `MaxLength`, formatting terminates, and an error is logged.

Converting FixedWidth documents to business objects

This section provides the following information on how the FixedWidth data handler converts FixedWidth documents to business objects:

- "FixedWidth string requirements"
- "Serialized-data processing" on page 137

FixedWidth string requirements

When converting a string to an business object, the FixedWidth data handler makes the following assumptions:

- The business object name appears as the first field in the data.
- The verb appears as the second field in the data.
- All attributes are given in the order that they appear in the business object definition.
- The `ObjectEventId` attribute is present in each business object. An entry for `ObjectEventId` is always required for a business object, even when it has the value `CxIgnore`, because the data handler uses it to distinguish between instances of a business object at runtime.

The format for a fixed-width string is as follows:

```
[Bus_Obj_Name<blank_space_padding_for_size>]
[Verb<blank_space_padding_for_size>]
[Attr1<blank_space_padding_for_size>]
[Attr2...<blank_space_padding_for_size>]
[Number-of-child-object_instances<blank_space_padding_for_size>]
[Child_Object_Name<blank_space_padding_for_size>]
[Child_Object_Verb<blank_space_padding_for_size>]
[Child_Object_Attr1<blank_space_padding_for_size>]
[Child_Object_Attr2...<blank_space_padding_for_size>]
<EndBO:Bus_Obj_Name>
```

In this format, the first two fields (`Bus_Obj_Name` and `Verb`) are padded to create fields of a length specified by the `BONameSize` and `BOVerbSize` attributes in the

FixedWidth child meta-object. The subsequent attributes are padded to create fields of a length specified in the MaxLength property for each business object attribute.

A field used with the FixedWidth data handler must be at least eight characters long if CxIgnore is a possible value for that attribute. If an attribute is guaranteed not to have a CxIgnore value, MaxLength can be less than eight characters long.

When a connector reads a file in fixed-width format, the CxIgnore and CxBlank meta-object attributes must be configured to generate the desired values. These strings affect the minimum MaxLength attributes. The minimum value for MaxLength must accommodate both.

Serialized-data processing

The FixedWidth data handler processes a FixedWidth document into a business object as follows:

1. The data handler creates a business object to contain the data. The type of business object is either passed into the conversion method by the connector, or the data handler extracts the name of the business object from the first field of the string. If the type parameter and the content in the data do not match, the data handler generates an error.
2. The data handler sets the verb in the business object. The data handler assumes that the verb for the top-level business object is in the second field in the data.
3. The data handler determines if there are any child meta-objects (those whose names are listed in the cw_mo_ tag of the business object application-specific information). The data handler does *not* perform the processing to populate these attributes of the business object. For more information about the cw_mo_ tag, see "Implementing conversion from a business object" on page 177.
4. The data handler looks for the meta-object attribute named OmitObjectEventId. If this is set to true, the data handler does not perform the processing to populate the ObjectEventId attribute.
5. To set the remaining business object attributes, the data handler parses the data based on the MaxLength of each attribute as specified in the business object definition. It extracts attribute values from the data and populates the values of the simple attributes in the business object.

   The data handler processes array attributes as follows:

   - If the attribute is a single-cardinality attribute, the data handler recursively parses the attribute list, sets the attribute values, and adds the child business object to the parent business object.
   - If the attribute is a multiple cardinality array, the data handler recursively parses the attributes in each child attribute list, and adds the child business object to the parent business object.

Chapter 7. Delimited data handler

The Delimited data handler converts business objects to delimited-format strings and streams, and from delimited-formatted strings and streams to business objects. This chapter describes how the Delimited data handler processes delimited data and how to define business objects to be processed by the data handler. You can use this information as a guide to modifying existing business objects or implementing new business objects that conform to the requirements of the data handler. This chapter also discusses how to configure the Delimited data handler. This chapter contains the following sections:

Note: The Delimited data handler is one of the base data handlers contained in the `CwDataHandler.jar` file. For information on how to install this data handler and where to store its source code, see Chapter 2, "Installing and configuring data handlers," on page 21.

Overview

The Delimited data handler is a data-conversion module whose primary role is to convert business objects to and from delimited-formatted strings or streams. A delimited-formatted string or stream is serialized data with the `text/delimited` MIME type. The data handler parses text data based on a specified delimiter that separates the individual fields of a business object's data. This type of data conversion is used primarily where the efficiency of machine reading is most important.

The default top-level connector meta-object (`MO_DataHandler_Default`) supports the `text/delimited` MIME type. Therefore, a connector that is configured to use the `MO_DataHandler_Default` meta-object can call the Delimited data handler. If InterChange Server is your integration broker and an access client must be able to call this data handler, you must modify the top-level server meta-object (`MO_Server_DataHandler`) to support the `text/delimited` MIME type. For more information, see "Modifying the top-level meta-object" on page 188.

Features of the Delimited data handler

The Delimited data handler allows you to set the following strings:

- The delimiter—The data handler uses a delimiter to separate the different fields in the delimited data. You can set the `Delimiter` meta-object attribute to the desired delimiter for your data. By default, the data handler uses a tilde (~) as the delimiter is size of the attribute property `MaxLength` to determine how to read and write data. `MaxLength` is an business object attribute property that defines the maximum number of characters of the attribute value, including padding to allow for right-justified or left-justified text. `MaxLength` is read from the definition of the business object in the repository. Accordingly, the main requirement for a business object is that the `MaxLength` for each string attribute is set appropriately.

- The escape string—The data handler uses the escape string to configure a string to escape the delimiter and escape strings. The escape string allows the attribute value data to contain delimiter-like and escape-like strings. You can set the `Escape` meta-object attribute to configure the escape string. By default, the data handler uses the backslash character (\) as the escape string.

Business object and string processing

The Delimited data handler performs the operations listed in Table 60.

Table 60. Data operations for the Delimited data handler

Data-handler operation	For more information
Receives a business object from the caller, converts the business object to a Delimited string or stream, and passes the serialized data to the caller.	"Converting business objects to delimited data" on page 143
Receives a string or stream from the caller, builds a business object, and returns the business object to the caller. Receives a Delimited string or stream from the caller, builds a business object, and passes the business object to the caller.	"Converting delimited data to business objects" on page 144

Configuring the Delimited data handler

To configure the Delimited data handler, take the following steps:

- Enter the appropriate values for the attributes of the Delimited child meta-object.
- Create or modify business object definitions so that they support the data handler.

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

Configuring the Delimited child meta-object

To configure a Delimited data handler, you must ensure that its configuration information is provided in the Delimited child meta-object. For the Delimited data handler, IBM delivers the `MO_DataHandler_DefaultDelimitedConfig` child meta-object. Each attribute in this meta-object defines a configuration property for the Delimited data handler. Table 61 describes the attributes in this child meta-object.

Table 61. Child meta-object attributes for the Delimited data handler

Meta-object attribute name	Meaning	Delivered default value
`ClassName`	Name of the data handler class to load for use with the specified MIME type. The top-level data-handler meta-object has an attribute whose name matches the specified MIME type and whose type is the Delimited child meta-object (described by Table 61).	`com.crossworlds.DataHandlers.text.delimited`
`CxBlank`	Establishes the equivalent value in the Delimited data for the special business object attribute value, Blank (the `CxBlank` constant). For more information, see "CxBlank" on page 142.	`CxBlank` constant (blank space)
`CxIgnore`	Establishes the equivalent value in the Delimited data for the special business object attribute value, Ignore (the `CxIgnore` constant). For more information, see "CxIgnore" on page 142.	`CxIgnore` constant (empty string)

Table 61. Child meta-object attributes for the Delimited data handler (continued)

Meta-object attribute name	Meaning	Delivered default value
`Delimiter`	String used to separate the values in business object attributes when writing business object data to files, or that is assumed to separate fields of data that correspond to attributes when converting a file to a business object. This value can contain multiple characters.	~ (tilde)
`DummyKey`	Key attribute required by the business integration system.	1
`Escape`	String used to escape the delimiter and escape characters if they occur in a business object attribute value. This value can only be one character in length.	\ (backslash)
`OmitObjectEventId`	Boolean value to determine whether or not to include `ObjectEventId` data in business object-to-String and String-to-business object conversions.	`false`
`ObjectEventId`	Placeholder attribute required by the business integration system.	none

The "Delivered default value" column in Table 61 lists the value in the Default Value property for the corresponding attribute in the delivered business object. You must examine your environment and set the Default Value properties of those attributes to the appropriate values for your system and your Delimited documents. You must make sure that at least the `ClassName` attribute has a default value.

Note: Use Business Object Designer to modify business object definitions.

Business object requirements

The Delimited data handler makes assumptions about the business objects that it handles. Therefore, when you create a business object for conversion using the Delimited data handler, follow these rules:

- Make sure that every business object attribute has a name to ensure that the Delimited data handler can properly convert the data.
- Make sure that the `ObjectEventId` attribute is included in every business object at all levels of a business object hierarchy. Business Object Designer does this automatically when it saves a business object definition, but you should confirm that the requirement is met.

The `Delimiter` attribute in the child meta-object configures the delimiter used to separate attribute fields. The default value is a tilde (~).

You can set the child meta-object attribute `Escape` to configure a string to escape the delimiter and escape strings. The escape string allows the attribute value data to contain delimiter-like and escape-like strings.

Business object structure

There are no requirements regarding the structure of the business objects for the Delimited data handler. The data handler can process any business object.

The business objects that the data handler processes can have any name allowed by the integration broker.

Business object attribute properties

Business object architecture contains various properties that apply to attributes. Table 62 describes how the Delimited data handler interprets several of these properties and describes how to set the properties when modifying a business object.

Table 62. Attribute properties for business objects converted using the Delimited data handler

Property name	Description
Name	Every business object attribute must have a unique name.
Type	Each business object attribute must have a type, such as Integer, String, or the type of a contained child business object.
Key	Not used by the Delimited data handler.
MaxLength	Not used by the Delimited data handler.
Foreign Key	Not used by the Delimited data handler.
Required	Not used by the Delimited data handler.
Default Value	Not used by the Delimited data handler.
Cardinality	Supports cardinality 1 and cardinality n objects.

Attributes in business objects can have the special values of `CxIgnore` or `CxBlank`. The Delimited data handler takes special processing steps when attributes have these values, as described in the following sections.

CxIgnore: The `CxIgnore` meta-object attribute establishes the equivalent value in a Delimited data for the `Ignore` attribute value (the `CxIgnore` constant). By default, the `CxIgnore` meta-object attribute is set to the value of the `CxIgnore` constant. The data handler uses the `CxIgnore` meta-object attribute as follows:

- When converting *from* a business object, the Delimited data handler writes the value of the `CxIgnore` meta-object attribute (the value configured for its Default Value property) to the Delimited data whenever it encounters a business object attribute with the `CxIgnore` constant as its attribute value.
- When converting *to* a business object, the Delimited data handler assigns the `CxIgnore` constant to the business object attribute's value whenever it encounters the value of the `CxIgnore` meta-object attribute (the value configured for its Default Value property) in the Delimited data.

Note: Business objects must have at least one primary key that does *not* contain the value `CxIgnore` at runtime.

CxBlank: The `CxBlank` meta-object attribute establishes the equivalent value in a Delimited data for the `Blank` attribute value (the `CxBlank` constant). By default, the `CxBlank` meta-object attribute is set to the value of the `CxBlank` constant. The data handler uses the `CxBlank` meta-object attribute as follows:

- When converting *from* a business object, the Delimited data handler writes the value of the `CxBlank` meta-object attribute (the value configured for the Default Value property) to the Delimited data whenever it encounters a business object attribute with the `CxBlank` constant as its attribute value.
- When converting *to* a business object, the Delimited data handler assigns the `CxBlank` constant to the business object attribute's value whenever it encounters the value of the `CxBlank` meta-object attribute (the value configured for its Default Value property) in the Delimited data.

Note: Business objects must have at least one primary key that does *not* contain the value `CxBlank` at runtime.

Business object application-specific information

The Delimited data handler does not require any application-specific information in business objects or their attributes. The data handler does, however, check for the existence of the cw_mo_ tag, which a business object might use to indicate any child meta-object that the connector uses. The data handler ignores any attribute identified by the cw_mo_ tag in the application-specific information of the business object. For more information about the cw_mo_ tag, see "Implementing conversion from a business object" on page 177.

Using existing business object definitions

The Delimited data handler can convert any business object to a Delimited string as long as the business object delivers data in a form that complies with the requirements of the data handler. The single requirement of the Delimited data handler is that if the data handler must read in a delimited file, that each individual field is separated by the configured delimiter.

Although existing business objects that meet this requirement can be converted by the Delimited data handler, a good practice is to create your own business objects for each type of data to be processed. If you use a sample business object, or a business object developed to support the same application in another implementation, be sure to modify the definition as necessary to include only the attributes required for the implementation for which you are developing.

Therefore, to convert existing business objects to a form that closely corresponds to your data, modify the business object to provide only the data required by the application and the information required by the data handler. To adapt existing business objects for use with the Delimited data handler, do the following:

1. Perform a functional analysis of the target application, and compare the results to existing business objects to determine the required fields of a business object definition.
2. Use Business Object Designer to add or delete attributes from the business object definition as needed.

Converting business objects to delimited data

To convert a business object to a string, the Delimited data handler loops through the attributes of a business object in sequential order. It generates Delimited formatting recursively in the order in which attributes appear in the business object and its children. The name of the business object is passed as an argument to the conversion method.

The Delimited data handler processes business objects into delimited data as follows:

1. The data handler creates a string to contain the data in the business object.
2. The data handler adds the business object name as the first token in the string and adds the verb as the second a token in the string.
3. The data handler examines the application-specific information in the business object definition to determine if there are any child meta-objects (those whose names are listed in the cw_mo_ tag of the business object application-specific information). The data handler does *not* include these attributes in the delimited data. For more information about the cw_mo_ tag, see "Implementing conversion from a business object" on page 177.

4. The data handler looks for the meta-object attribute named `OmitObjectEventId`. If this is set to `true`, the data handler does not include `ObjectEventId` data of the business object in the delimited data.
5. The data handler loops through the remaining business object attributes in order, adding values for each simple attribute to the string and adding the configured delimiter after each attribute. For container attributes, the data handler does the following:
   - If the attribute is a cardinality 1 container, the data handler adds the attribute count to the string, and then recursively processes the child business object to add values for each attribute.
   - If the attribute is a cardinality n container, the data handler adds the count of the child objects in the container to the string, and then recursively processes each child business object, adding values for each attribute to the string.
6. When the data handler completes the conversion, it returns the serialized data to the caller. The data handler returns the data in the form (String or InputStream) requested by the caller.

The format that the data handler generates conforms to the following pattern:

```
Bus_Obj_Name<delimiter>Verb<delimiter>Attr1<delimiter>Attr2<delimiter>
Number_of_child_object_instances<delimiter>Child_Object_Name<delimiter>Verb
<delimiter>Attr1<delimiter>Attr2<EndBO:Bus_Obj_Name>
```

An escape string is appended in front of any delimiter-like string in an attribute value. The escape string is configured using the `Escape` attribute of the child meta-object.

Converting delimited data to business objects

This section provides the following information on how the Delimited data handler converts Delimited data to business objects:

- "Delimited string requirements"
- "Serialized-data processing" on page 145

Delimited string requirements

When converting a string or stream, the Delimited data handler makes the following assumptions:

- The data contains the delimiter specified in the `Delimiter` meta-object attribute.
- The business object name appears in the first field in the data.
- The verb appears as the second field in the data.
- Attributes are in the order that they appear in the business object definition.
- All objects in a child container are of the same type.
- The data contains a token that represents the number of child objects in each cardinality `n` container.
- The `ObjectEventId` attribute is present in each business object. An entry for `ObjectEventId` is always required for a business object, even when it has the value `CxIgnore`, because the data handler uses it to distinguish between instances of a business object at runtime.

If you have more than one business object in the data, make sure you do not introduce any new characters (such as a space, a tab, a new line, or a carriage return) between them.

When the Delimited data handler reads a file in Delimited format, it takes the following special processing steps to assign to a business object attribute the CxIgnore or CxBlank attribute value:

- It assigns the CxIgnore constant (null) as the corresponding attribute value whenever it encounters any of the following conditions in the Delimited data:
  - The value of the CxIgnore meta-object attribute (the value configured for its Default Value property)
  - The value of an empty string (" ")
- It assigns the CxBlank constant as the attribute value only when the CxBlank meta-object attribute is configured and it encounters this configured value in the corresponding Delimited data.

Note: Make sure the escape string and the delimiter have different values as configured by the Escape and Delimiter meta-object attributes in the Delimiter data handler child meta-object.

The following line shows an example of a string in Delimited format. The syntax is:

```
Bus_Obj_Name<delimiter>Verb<delimiter>Attr1<delimiter>Attr2<delimiter>
Number_of_child_object_instances<delimiter>Child_Object_Name<delimiter>
Verb<delimiter>Attr1<delimiter>Attr2<EndBO:Bus_Obj_Name>
```

The following sample uses a tilde (~) delimiter:

```
Customer~Create~p1~p2~p3~1~CustomerAddress~Create~q1~q2~q3~q4~q5~q6~q7~q8~q9~q10~3~
PhoneInfo~Create~r1~r2~r3~r4~r5~r6~r7~PhoneInfo~Create~r1~r2~r3~r4~r5~r6~r7~
PhoneInfo~Create~r1~r2~r3~r4~r5~r6~r7
```

Serialized-data processing

The Delimited data handler processes delimited data into a business object as follows:

1. The data handler gets the business object name from the first token in the data and creates a business object to contain the data.
2. The data handler sets the verb in the business object. The data handler assumes that the verb for the top-level business object is in the second token in the delimited data. Note that child business objects may not have verbs set.
3. The data handler determines if there are any child meta-objects (those whose names are listed in the cw_mo_ tag of the business object application-specific information). The data handler does *not* perform the processing to populate these attributes of the business object. For more information about the cw_mo_ tag, see "Implementing conversion from a business object" on page 177.
4. The data handler looks for meta-object attribute named OmitObjectEventId. If this is set to true, the data handler does not perform the processing to populate the ObjectEventId attribute.
5. The data handler parses the data and populates the values of the remaining simple attributes in the business object with the token values from the data. The data handler processes container attributes as follows:
   - If the attribute is single-cardinality, the data handler recursively parses the attribute tokens in the string, sets the attribute values in the business object, and adds the child business object container to the parent business object.
   - If the attribute is multiple cardinality, the data handler recursively parses the attribute tokens for each child object, sets the attribute values in the child business object, and adds the child business object container to the parent business object.

Chapter 8. NameValue data handler

The NameValue data handler converts a business object to string or stream formatted in name-value pairs, and converts a string or stream formatted in name-value pairs to a business object. This chapter describes how the NameValue data handler processes NameValue data and describes how to define business objects to be processed by the NameValue data handler. You can use this information as a guide to implementing business objects that conform to the requirements of the NameValue data handler. This chapter also discusses how to configure the NameValue data handler. This chapter contains the following sections:

Note: The NameValue data handler is one of the base data handlers contained in the `CwDataHandler.jar` file. For information on how to install this data handler and where to store its source code, see Chapter 2, "Installing and configuring data handlers," on page 21.

Overview

The NameValue data handler is a data-conversion module whose primary role is to convert a business object to and from a string or stream formatted in name-value pairs. A NameValue-formatted string or stream is serialized data with the `text/namevalue` MIME type. The NameValue data handler can be used to generate a business object file for testing purposes.

The data handler parses serialized data based on named fields. For example, a text file for this data handler contains fields that identify the business object type (BusinessObject=BOname), verb (Verb=verbName), number of attributes (AttributeCount=numericValue), and attribute values (`AttributeName=`*`Value`*). The data handler uses the name-value information to parse the data.

The default top-level connector meta-object (`MO_DataHandler_Default`) supports the `text/namevalue` MIME type. Therefore, a connector that is configured to use the `MO_DataHandler_Default` meta-object can call the NameValue data handler. For an access client to be able to call this data handler when using the InterChange Server integration broker, you must modify the `MO_Server_DataHandler` meta-object to support the `text/namevalue` MIME type. For more information, see "Modifying the top-level meta-object" on page 188.

The NameValue data handler performs the operations listed in Table 63.

Table 63. Data operations for the NameValue data handler

Data-handler operation	For more information
Receives a business object from the caller, converts the business object into a NameValue string or stream, and passes the serialized data to the caller.	"Converting business objects to NameValue data" on page 151

Table 63. Data operations for the NameValue data handler (continued)

Data-handler operation	For more information
Receives a string or stream from the caller, builds a business object, and returns the business object to the caller.	"Converting NameValue data to business objects" on page 152

Configuring the NameValue data handler

To configure the NameValue data handler, take the following steps:

- Enter the appropriate values for the attributes of the NameValue child meta-object.
- Create or modify business object definitions so that they support the data handler.

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

Configuring the NameValue child meta-object

To configure a NameValue data handler, you must ensure that its configuration information is provided in the NameValue child meta-object. For the NameValue data handler, IBM delivers the `MO_DataHandler_DefaultNameValueConfig` child meta-object. Each attribute in this meta-object defines a configuration property for the NameValue data handler.Table 64 describes the attributes for this child meta-object.

Table 64. Child meta-object attributes for the NameValue data handler

Meta-object attribute name	Description	Delivered default value
`ClassName`	Name of the data handler class to load for use with the specified MIME type. The top-level data-handler meta-object has an attribute whose name matches the specified MIME type and whose type is the NameValue child meta-object (described by Table 64).	`com.crossworlds.DataHandlers.text.namevalue`
`CxBlank`	Establishes the equivalent value in the NameValue data for the special business object attribute value, Blank (the `CxBlank` constant). For more information, see "CxBlank" on page 150.	`CxBlank` constant
`CxBlankValue`	*This attribute is deprecated.* Use the `CxBlank` meta-object attribute (above) to tell the data handler how the NameValue data represents the `CxBlank` attribute value.	blank space
`CxIgnore`	Establishes the equivalent value in the NameValue data for the special business object attribute value, Ignore (the `CxIgnore` constant). For more information, see "CxIgnore" on page 149.	`CxIgnore` constant
`DefaultVerb`	Business object verb	`Create`
`DummyKey`	Key attribute required by the business integration system.	`1`
`SkipCxIgnore`	During request processing, the processing of the special attribute value `CxIgnore` is based on the meta-object attribute `SkipCxIgnore`. For more information, see "CxIgnore" on page 149.	`false`
`ValidateAttrCount`	Determines whether the data handler looks for (or adds to the output string) a token that contains a count of attributes in the business object data.	`true`
`ObjectEventId`	Placeholder not used by the data handler but required by the business integration system.	none

The "Delivered default value" column in Table 64 lists the value in the Default Value property for the corresponding attribute in the delivered business object. You must examine your environment and set the Default Value properties of those attributes to the appropriate values for your system and your name-value pair-formatted documents.

Note: Use Business Object Designer to modify business object definitions.

Business object requirements

The NameValue data handler makes assumptions about the business objects that it handles. Therefore, when you pass a business object for conversion with the NameValue data handler, follow these rules:

- Make sure that every business object attribute has a `Name` property. This ensures that the NameValue data handler can properly process the conversion of data from a business object to a NameValue format, and a NameValue format to a business object.
- Make sure that the `ObjectEventId` attribute is included in every business object at all levels of a business object hierarchy. Business Object Designer does this automatically when it saves a business object definition, but you should confirm that the requirement is met.

Business object structure

There are no requirements regarding the structure of the business objects for the NameValue data handler. The data handler can process any business object.

The business objects that the data handler processes can have any name allowed by the business integration system.

Business object attribute properties

Business object architecture contains various properties that apply to attributes. Table 65 describes how the NameValue data handler interprets several of these properties and describes how to set the properties when modifying a business object definition.

Table 65. Attribute properties for business objects converted using the NameValue data handler

Property name	Description
Name	Each business object attribute must have a unique name.
Type	Each business object attribute must have a type, such as integer, String, or the type of a contained child business object. All simple attributes must be of type String.
Key	Not used by the NameValue data handler.
MaxLength	Not used by the NameValue data handler.
Foreign Key	Not used by the NameValue data handler.
Required	Not used by the NameValue data handler.
Default Value	Not used by the NameValue data handler.
Cardinality	Supports cardinality 1 and cardinality n objects.

Attributes in business objects can have the special values of `CxIgnore` or `CxBlank`. The NameValue data handler takes special processing steps when attributes have these values, as described in the following sections.

CxIgnore: The `CxIgnore` meta-object attribute establishes the equivalent value in a NameValue data for the `Ignore` attribute value (the `CxIgnore` constant). By default,

the CxIgnore meta-object attribute is set to the value of the CxIgnore constant. The data handler uses the CxIgnore meta-object attribute as follows:

- When converting *from* a business object, the NameValue data handler writes the value of the CxIgnore meta-object attribute (the value configured for its Default Value property) to the NameValue data whenever it encounters a business object attribute with the CxIgnore constant as its attribute value.
- When converting *to* a business object, the NameValue data handler assigns the CxIgnore constant to the business object attribute's value whenever it encounters any of the following conditions in the NameValue data:
  - The value of the CxIgnore meta-object attribute (the value configured for its Default Value property)
  - The value of an empty string
  - No corresponding value

Note: Business objects must have at least one primary key that does *not* contain the value CxIgnore at runtime.

You can configure how the NameValue data handler processes attributes with an attribute value of CxIgnore. For example:

- You can configure whether you want the data handler to process attributes with a CxIgnore value during request processing or ignore them.
- You can decide not to make an entry for attributes with a CxIgnore value so that the connector can process the business object data correctly during event notification. This is helpful when creating a dummy file for an object type.

During request processing, the data handler is creating a serialized version of the business object. At this time, the processing of the special attribute value CxIgnore is based on the child meta-object attribute SkipCxIgnore, as follows:

- When SkipCxIgnore is set to false, the data handler writes the value of this CxIgnore meta-object attribute to the NameValue data whenever it encounters a business object attribute with the CxIgnore constant as the attribute's value.
- If SkipCxIgnore is set to true, the data handler ignores all attributes with a value of CxIgnore and does *not* generate any NameValue data for them.

Note: With SkipCxIgnore set to true, the NameValue data handler is *not* bidirectional; that is, it cannot perform string-to-business-object conversions on strings that it has generated during business-object-to-string conversions.

CxBlank: The CxBlank meta-object attribute establishes the equivalent value in a NameValue data for the Blank attribute value (the CxBlank constant). By default, the CxBlank meta-object attribute is set to the value of the CxBlank constant. The data handler uses the CxBlank meta-object attribute as follows:

- When converting *from* a business object, the NameValue data handler writes the value of the CxBlank meta-object attribute (the value configured for the Default Value property) to the NameValue data whenever it encounters a business object attribute with the CxBlank constant as its attribute value.
- When converting *to* a business object, the NameValue data handler assigns the CxBlank constant to the business object attribute's value whenever it encounters the value of the CxBlank meta-object attribute (the value configured for its Default Value property) in the NameValue data.

Note: Business objects must have at least one primary key that does *not* contain the value CxBlank at runtime.

Business object application-specific information

The NameValue data handler does not require any application-specific information in business objects or their attributes. The data handler does, however, check for the existence of the cw_mo_ tag, which a business object might use to indicate any child meta-object that the connector uses. The data handler ignores any attribute identified by the cw_mo_ tag in the application-specific information of the business object. For more information about the cw_mo_ tag, see "Implementing conversion from a business object" on page 177.

Using existing business object definitions

The NameValue data handler can convert any business object to NameValue serialized data as long as the business object delivers data in a form that complies with the requirements of the data handler. The NameValue data handler requires each piece of data to have a name that identifies it, such as BusinessObject=Customer, Verb=Create, and CustomerName=JDoe. Because attributes must have such a name, they can be used with the NameValue data handler.

Although existing business objects that meet this requirement can be converted by the NameValue data handler, a good practice is to create your own business objects for each type of data to be processed. If you use a sample business object, or a business object developed to support the same application in another implementation, be sure to modify the definition as necessary to include only the attributes required for the implementation for which you are developing.

Therefore, to convert existing business objects to a form that closely corresponds to your data, modify the business object to provide only the data required by the application and the information required by the data handler. To adapt existing business objects for use with the NameValue data handler, do the following:

1. Perform a functional analysis of the target application, and compare the results to existing business objects to determine the required fields of a business object definition.
2. Use Business Object Designer to add or delete attributes from the business object definition as needed.

Converting business objects to NameValue data

To convert a business object to a string or stream, the NameValue data handler loops through the attributes of a business object in sequential order. It generates name-value pairs recursively in the order in which attributes appear in the business object and its children. The name of the business object is passed as an argument to the conversion method.

The NameValue data handler processes business objects into NameValue data as follows:

1. The data handler creates a string to contain the data in the business object.
2. To specify the business object name, the data handler adds BusinessObject=*Name* to the string.
3. To specify the verb, the data handler adds Verb=*Verb* to the string.
4. If the meta-object attribute ValidateAttrCount is set to true, the data handler adds AttributeCount=*Count* to the string. This name-value pair specifies the number of the attributes in the business object data.
5. The data handler examines the application-specific information in the business object definition to determine if there are any child meta-objects (those whose names are listed in the cw_mo_ tag of the business object application-specific

information). The data handler does *not* include these attributes in the NameValue data. For more information about the cw_mo_ tag, see "Implementing conversion from a business object" on page 177.

6. The data handler loops through the remaining business object attributes in order, adding name-value pairs for each simple attribute to the string. For container attributes, the data handler does the following:
   - If the attribute is a cardinality 1 container, the data handler adds the attribute name and a count of 1 to the string, and then recursively processes the child business object to add name-value pairs for each attribute to the string.
   - If the attribute is a cardinality n container, the data handler adds the attribute name and the number of child objects in the container to the string, and then recursively processes each child business object, adding name-value pairs for each attribute to the string.
7. When the data handler completes the conversion, it returns the serialized data to the caller. The data handler returns the data in the form (String or InputStream) requested by the caller.

If the child meta-object attribute ValidateAttrCount is true, the data handler adds a token that contains a count of the attributes in the business object to the output data. The data handler adds carriage returns to output data; the end result looks like Figure 36 on 153.

Converting NameValue data to business objects

This section provides the following information on how the NameValue data handler converts strings or streams formatted in name-value pairs to a business object:

- "NameValue string requirements"
- "Serialized-data processing" on page 154

NameValue string requirements

The NameValue data handler makes the following assumptions about serialized data:

- The business object name appears in the first name-value pair.
- The verb appears in the second name-value pair.
- The data contains a token that represents the number of instances of child objects for each child that is contained in a business object.
- The ObjectEventId attribute is present in each business object.

A token representing the attribute count is optional. If the child meta-object attribute ValidateAttrCount is true, the data handler looks for a token that contains a count of the attributes in the business object. If the attribute count is specified, it must accurately reflect the number of attributes in the business object definition.

When the NameValue data handler reads a file in name-value format, it takes the following special processing steps to assign to a business object attribute the CxIgnore or CxBlank attribute value:

- The data handler assigns the CxIgnore constant (null) as the corresponding attribute value whenever it encounters any of the following conditions in the NameValue data:

  - The value of the CxIgnore meta-object attribute (the value configured for its Default Value property)
  - The value of an empty string (″ ″)
  - No corresponding value in the NameValue data for the business object attribute.
- The data handler assigns the CxBlank constant as the attribute value only when the CxBlank meta-object attribute is configured and it encounters this configured value in the corresponding NameValue data.

Figure 36 shows an example of serialized data in NameValue format.

```
BusinessObject=Customer
  Verb=Update
          AttributeCount=7
          CustomerID=103
          CustomerName=Thai Inc.
          Cust_Phone_Number=CxIgnore
          ProductName=GoodProduct
          Address=2
            BusinessObject=Address
              Verb=Update
              AttributeCount=3
              AddressID=105
              AddressLine=CxIgnore
              ObjectEventID=12345
            BusinessObject=Address
              Verb=Delete
              AttributeCount=3
              AddressID=106
              AddressLine=2758 Forest Avenue
              ObjectEventID=CxIgnore
          Item=1
            BusinessObject=Item
              Verb=Update
              ItemID=107
              ItemName=CxIgnore
              ObjectEventID=Obj_201
          ObjectEventID=SampleConnector_894927711_2
```

Figure 36. Example of NameValue data

In this example, entries indicate the following:

- BusinessObject is the name of the parent or child business object being processed.
- Verb is the type of request (for example, Create or Update) with which the parent or child business object is being sent.
- AttributeCount is the total number of attributes for the parent or child business object at that level.
- CustomerID, CustomerName, Cust_Phone_Number, and ProductName are the names of the attributes for the parent business object. Values for each parent business object attribute follow the attribute name.
- Address = 2 indicates that there are two instances of the Address child business object. Address is the attribute name that refers to the Address child business object in the parent object.
- Item = 1 indicates that the Item attribute contains a single instance of the Item business object.

- `AddressID` and `AddressLine` are the names of the attributes for the `Address` child business object. Values for each child business object attribute follow the attribute name.
- `ObjectEventID=Obj_201` is the system-generated ID for the child business object, `Item`.
- `ObjectEventID=SampleConnector_894927711_2` is the system-generated ID for the parent business object, `Customer`.

Serialized-data processing

The NameValue data handler converts strings or streams formatted in name-value pairs to a business object as follows:

1. The data handler creates a business object to contain the data in the string or stream.
2. The data handler sets the verb in the business object. The data handler assumes that the verb for the top-level business object is in the second name-value pair in the data. Note that child business objects may not have verbs set.
3. If the `ValidateAttrCount` child meta-object attribute is set to `true`, the data handler validates that the number of attributes in the file matches the number of attributes in the business object definition.
4. The data handler parses the serialized data.
   - It first determines if there are any child meta-objects (those whose names are listed in the `cw_mo_` tag of the business object application-specific information). The data handler does *not* perform the processing to populate these attributes of the business object. For more information about the `cw_mo_` tag, see "Implementing conversion from a business object" on page 177.
   - It populates the values of the remaining simple attributes in the business object. The data handler processes container attributes as follows:

     If the attribute is single cardinality, the data handler recursively parses the attributes in the attribute list and adds the child business object container to the parent business object.

     If the attribute is multiple cardinality, the data handler recursively parses the attributes in each child attribute list, and adds the child business object container to the parent business object.

You can specify the attributes in the serialized data in any order for the string-to-business object conversion because the data handler does a name and value association.

Chapter 9. Binary host data handler

The IBM WebSphere Business Integration Data Handler for COBOL records, called the binary host data handler, converts business objects to COBOL records and COBOL records to business objects. The chapter describes how the binary host data handler processes COBOL records and how to define business objects to be processed by the data handler. You can use the information as a guide to implementing business objects that conform to the requirements of the binary host data handler. This chapter contains the following sections:

Overview

The binary host data handler is a data-conversion module whose primary role is to convert business objects to and from host system COBOL records.

In addition to converting host records into business objects and vice versa, the binary host data handler parses COBOL records containing the Packed Decimal (PD) and Double Byte Character Set (DBCS) data available in the form of bytes, along with the general ASCII/Text data.

The binary host data handler handles the COBOL datatypes described in Table 66.

Table 66. Datatypes supported by the binary host data handler

Data Type	PIC clause format
Alphabetic	A(n), AAA... n times
Numeric	9(n), 999... n times
Alphanumeric	X(n), XXX... n times
Packed decimal	9(n) with a COMP-3 clause
DBCS	G(n), GGG... n times

In addition to these data types, the binary host data handler also supports a number of other PICTURE clauses (for example, V, P, S, and others).

The data handler can be plugged into and used by the JText Connector Agent. The following section describe the binary host data handler in more detail. For more information about configuring and using the JText Connector Agent, see the *Adapter for Jtext User Guide*.

Processing COBOL records and business objects

COBOL records use a template, called a `copybook`, to define their structure. Figure 37 on page 156 shows the basic structure of a COBOL record, which is specified in the data division of a COBOL program.

```
Data Division.
          File Section.
                FD Customer-File
                Record Contains 50 Characters.
                01 Customer-Record.
                   05 Customer-Name.
          10 Last-Name Pic x(17).
          10 Filler Pic x.
          10 Initials Pic xx.
       05 Part-Order.
          10 Part-Name Pic x(15).
          10 Part-Color Pic x(15).
  Working-Storage Section.
    01 Orig-Customer-Name.
       05 Surname Pic x(17).
       05 Initials Pic x(3).
    01 Inventory-Part-Name Pic x(15).
    01 Inventory-Part-Color Pic x(15).
```

Figure 37. COBOL record structure

The file section of the data division denotes the structure of the record as specified in the file, whereas the working-storage section denotes the handling of the data within the COBOL program.

The COBOL copybook corresponds to the file section of the data division. The copybook corresponding to the record with the above structure appears in Figure 38.

```
01  CUSTOMER-RECORD.
  05  CUSTOMER-NAME
        10 LAST-NAME          PIC X(17)      USAGE DISPLAY-1
        10 FILLER                PIC X
        10 INITIALS              PIC XX
  05  PART-ORDER
        10 PART-NAME          PIC X (15)      USAGE DISPLAY-1
        10 PART-COLOR         PIC X (15)
```

Figure 38. Sample copybook

Just as a copybook describes the structure of the COBOL record, the business object definition describes the structure of the business object. The binary host data handler uses business object definitions when it converts between business objects and COBOL records. It determines how to perform the conversion using the structure of the business object definition and its application-specific information.

A properly-constructed business object definition ensures that the data handler can correctly convert a business object to a COBOL record and a COBOL record to a business object. Before the binary host data handler can perform a conversion between the COBOL record and the business object, it must be able to locate the associated business object definition.

Use of the binary host data handler to convert a COBOL record to a business object or a business object to a COBOL record requires that the following steps occur.

Table 67. Using the binary host data handler

Step	For more information
1. Business object definitions that describe the COBOL record and business-object structure must exist and be available to the binary host data handler when it executes.	"Business object definitions for COBOL records" on page 161 "Creating business object definitions for COBOL records" on page 161
2. The binary host data handler must be configured for your environment.	"Configuring the binary host data handler" on page 159
3. The binary host data handler must be called from the Jtext connector to perform the appropriate data operation:	
a) Data operation: receive a business object from the caller, convert the business object to a COBOL record, and pass the COBOL record to the caller.	"Converting business objects to COBOL records" on page 162
b) Data operation: Receive a COBOL record from the caller and use binary host components to build a business object based on the supplied binary datatype. Then return the business object to the caller.	"Converting COBOL records to business objects" on page 162

Binary host data handler limitations

The binary host data handler has been created specifically for use with certain binary data formats, with COBOL records, and with Jtext connector in particular. Limitations on its capibilities in a business integration system are as follows:

- The data handler is designed to support only one record schema at a time. It doesn't support multiple schemas. Generally, however, binary host applications deal with a single schema at a time.
- Currently, there is no ODA for converting a COBOL copybook to a corresponding business object. Users must create the business object definitions for the COBOL records themselves before using the binary host data handler at runtime.
- The binary host data handler can only be used by an adapter configured to work with binary data processing.
- The data handler has been optimized for use on MVS, z/OS, AS/400, and other variants of these.

Binary host components

Internally, the binary host data handler uses different components to handle the various types of data in a COBOL record. Figure 39 on page 158 shows these components.

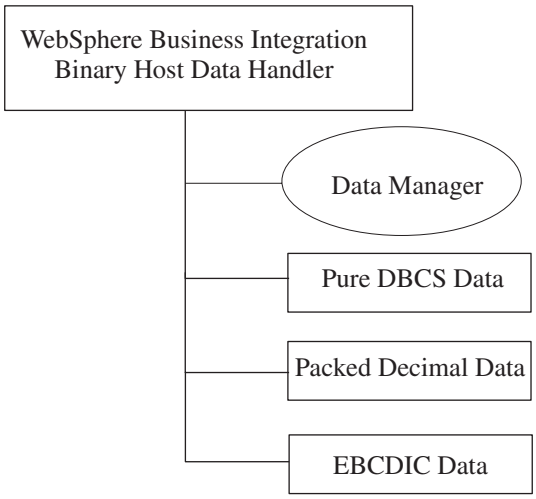

Figure 39. Binary host data handler components

These components are described in detail in the following sections.

Pure DBCS data

Any attribute with ASI specifying a COBOL PICTURE Clause of G(x) and the datatype as DBCS will be treated as DBCS data by the binary host data handler. The Pure DBCS data component of the data handler will process DBCS data retrieved from the mainframe as follows :

- Adding shift-in, shift-out characters to obtain a string
- Decoding this string with the encoding format specified by the user (Cp930, by default), to get a Unicode Java String object.

The minimum length of any DBCS data that can be processed is one character, i.e. 2 bytes. The COBOL Compiler restricts the maximum length. DBCS data is always right padded with double-byte blank/space characters.

EBCDIC data

The binary host data handler recognizes EBCDIC data with application specific information in the BO definition that contains any of the alphabetic COBOL PICTURE Clauses (A or X), and the datatype specified as EBCDIC. The EBCDIC component of the data handler decodes EBCDIC data with the encoding format specified by the user in the 'BinaryEncoding' property of the MO_ DataHandler_DefaultBinaryHostConfig (Cp930, by default) to get a Unicode Java String object.

The minimum limiting length for any EBCDIC data is one character, corresponding to one byte. The COBOL Compiler restricts the maximum length. This data is always right padded with blank/space characters.

Packed decimal data

Packed Decimal data is recognized by the binary host data handler through application specific information in the business object definition that contains any of the numeric PIC clauses (9 or X), with a COMP-3 clause, and the datatype specified as Packed Decimal.

The base limit on Packed Decimal data is one digit. The digit has to be a signed integer, and cannot represent a float value. The Packed decimal data component

does not supercede the maximum value that a packed decimal field in COBOL can hold, specified as 15 digits by S/390 architecture. Packed decimal data is always left padded with zeroes.

Note: The example input 999 (base 10) corresponds to the hex value: 03D7 (base 16). Hence, if it is not packed, it will be stored as 000003D7. When packed, each digit gets stored in a nibble, and hence, the value becomes 999C (base 16).

The rightmost nibble denotes the sign of the packed decimal. A rightmost nibble of A, C, E, or F implies positive, and B or D implies negative.

Configuring the binary host data handler

To configure the binary host data handler for use with a connector, take the following steps:

- Enter the appropriate values for the attributes of the connector configuration object.
- Enter the appropriate values for the attributes of the binary host child meta-object.

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

Note: To use the binary host data handler, you must also create or modify business object definitions so that they support the data handler. For more information, see "Business object definitions for EDI documents" on page 94.

Note: The binary host data handler can only be used with an adapter configured to work with binary processing. The JText adapter is one such adapter.

Configuring the JText connector configuration object

The following attributes need to be set in the MO_JTextConnector_Default file, which is the connnector configuration object.

Table 68. Attributes of the MO_JTextConnector_Default

Attribute name	Description
`EventDataHandler`	Sets the DataHandler for Event Processing scenario. Must have the MO_DataHandler_DefaultBinaryHostConfig selected from the drop box containing object types.
`OutputDataHandler`	Sets the DataHandler for Service Call scenario. Must have the MO_DataHandler_DefaultBinaryHostConfig selected from the drop box containing object types.
`DataProcessingMode`	Must always be ″binary″ for connecting to the binary host data handler. The binary host data handler throws an error if unspecified, or if specified as 'text'.
`FTPTransferType`	Must always be ″record″. The binary host data handler throws an error if unspecified, or if specified as 'file'.
FTPDataStructure	Must always be ″record″. The default value is ″file″.

Refer to the *Adapter for Jtext User Guide* for more information about how to configure these properties of the JText connector.

Configuring the binary host data handler child meta-object

MO_DataHandler_DefaultBinaryHostConfig object is the meta object that defines the properties of the binary host data handler that determine the transformation of binary data.

The EventDataHandler and OutputDataHandler attributes in the MO_JTextConnector_Default must be set to MO_DataHandler_ DefaultBinaryHostConfig to enable the binary host data handler.

After setting these attributes, the child attributes of MO_DataHandler_DefaultBinaryHostConfig have to be set, to customize the behavior of the binary host data handler in the event or request processing scenarios. Table 69 describes the attributes in this child meta-object.

Table 69. Child meta-object attributes for the binary host data handler

Attribute name	Description	Delivered default value	Required
`ClassName`	Name of the data handler class to load for processing the binary data. Note that the complete package structure needs to be specified.	`com.crossworlds. DataHandlers.binary`	Yes
`BODefinitionName`	Name of the BO template that the user has created to support the COBOL copybook.		Yes
`CxBlank`	Field used by the data handler to handle blanks. User need not specify anything here.	`CxBlank`	No
`CxIgnore`	Field used by the data handler to handle null value. User need not specify anything here.	`CxIgnore`	No
`DefaultVerb`	Specifies the default verb used by the data handler for its operations on application specific business objects (ASBOs)	`Create`	No
`BinaryEncoding`	Specifies the encoding which the binary host data handler uses to decode data obtained from MVS, or to encode data sent to MVS.		Yes
`DummyKey`	Key field; has a unique value for every instance.	1	Yes

The "Delivered default value" column in Table 32 lists the value in the Default Value property for the corresponding attribute in the delivered business object. You must examine your environment and set the Default Value properties of all the attributes to the appropriate values.

Note: Use Business Object Designer to modify business object definitions.

For more information about how to configure a data handler, see "Configuring data handlers" on page 24.

Business object definitions for COBOL records

To use the binary data handler, you must create or modify business object definitions so that they contain the metadata that the data handler requires and so that they include the fields that correspond to those in the COBOL record. This section provides the information you need to create business object definitions to work with the binary host data handler. In particular, it provides the following information:

- "Understanding binary host business object structure"
- "Creating business object definitions for COBOL records"

Understanding binary host business object structure

The binary host data handler is comprised of a manager that manages several components, each of which is capable of interpreting specific datatypes. These datatypes, Pure DBCS, Packed Decimal or EBCDIC, are described in "Binary host components" on page 157.

The user is expected to develop the BO Definition according to the copybook that corresponds to the event file record structure. See "Processing COBOL records and business objects" on page 155 for more information about the COBOL copybook format.

As a part of the application specific information (ASI) of the attributes, the user needs to specify certain parameters, the details and applicable values for which can be obtained from Table 70:

Table 70. ASBO definitions for COBOL records

Attribute Name	Description	Valid Values
Datatype	Specifies the data type category of the data. Has been included because the PIC clause alone is not enough for deciding the datatypes. This ASI is required and has to be specified for all the attributes.	DBCS ; PackedDecimal ; EBCDIC
PICClause	Specifies the PIC clause exactly as specified in the COBOL copybook, along with the length. This ASI is required and has to be specified for all the attributes.	X(5); G(8);
COMPClause	Specifies the COMP clause of a field, if exists. The packed decimal fields generally have a COMP-3 associated with them. This ASI is optional and not required for non-numeric data.	COMP-1, COMP-3

The binary host data handler interprets binary data based on the value of these parameters in the application specific information.

Creating business object definitions for COBOL records

This section describes how to manually create business object definitions to represent COBOL records. The business object definitions are based on `COBOL copybooks`, which provide data definitions for the COBOL records. Since there are not specific object discovery agents (ODAs) for converting COBOL copybooks to business object definitions, you must use the Business Object Designer to create the

business object definitions. The Business Object Designer lets you add or delete attributes from business object definitions, as well as edit attribute properties, as needed.

To define a business object based on a COBOL copybook:

1. Open the Business Object Designer and specify 'New BO'.

   You can also select File -> New... -> Business Object Name and leave the ASI blank.
2. Specify the record name as the BusinessObject Name. For example, specify CustomerRecord or Customer.
3. In the General tab of the BO Designer, delete the verbs Delete, Retrieve and Update by selecting them and pressing the Delete key. Retain only the Create verb.
4. Go to the Attributes tab and start create attributes for the BO Definition, based on the copybook fields. Create a new attribute above Object Event ID.

Converting business objects to COBOL records

To convert a business object to a COBOL record, the binary host data handler loops through the attributes of the top-level business object definition. It processes the attributes recursively, in the order in which they appear in the top-level business object, writing attribute values as the elements of the COBOL record.

The binary host data handler processes business objects into a COBOL record as follows:

1. The binary host data handler instantiates a binary byte array to hold the record data.
2. It retrieves the business object definition corresponding to the Request BO.
3. It parses the metadata stored in the business object definition (name, type of data, length in bytes)

   Processing of data involves the additional transformations:

Data Type	Processing
ASCII/EDBDIC	As is
Numeric (General)	As is
Packed Decimal	Unpack the bytes
DBCS	Decode using Cp930

4. The data handler populates the business object instance with processed bytes.
5. Finally, the data handler returns the populated business object back to the adapter.

Converting COBOL records to business objects

To convert a COBOL record to a business object, the binary host data handler loops through the attributes of the top-level business object definition. It obtains the name of the business object to create, then processes the attributes recursively, in the order in which attributes appear in the top-level business object and its children, assigning values from the binary record to the business object.

The steps the data handler takes to process the binary data into a business object are as follows:

1. The data handler instantiates the business object specified, choosing the business object definition from the metadata library.
2. It parses the metadata stored in the business object definition (name, type of data, length in bytes).
3. The data handler extracts the stream of bytes based on the length and datatype information, extract the stream of bytes.

   Processing of data involves the additional transformations:

Data Type	**Processing**
ASCII/EDBDIC	As is
Numeric (General)	As is
Packed Decimal	Unpack the bytes
DBCS	Decode using Cp930

4. It generates the binary stream of data with processed bytes.
5. Finally, the data handler returns the binary stream back to the adapter.

Part 2. Custom data handlers

Chapter 10. Creating a custom data handler

This chapter presents information on how to implement a custom data handler to use with a WebSphere business integration adapter or, if your integration broker is InterChange Server, an access client. As with IBM-delivered data handlers, a custom data handler converts a business object to a specific serialized data format, and converts serialized data in a specific format to a business object. This chapter contains the following sections:

Overview of the data-handler development process

To develop a custom data handler, you code the data handler source file and complete other tasks, such as developing a meta-object for the data handler. The task of creating a custom data handler includes the following general steps:

1. Design the data handler, based on the format of the serialized data and structure of the business objects it converts.
2. Create a class that extends the class:

   `com.crossworlds.DataHandlers.DataHandler`
3. Implement the abstract methods that convert data between specific data format and business objects.
4. Compile the class and add it to the `CustDataHandler.jar` file.
5. Create the data-handler meta-objects.
6. Develop business object definitions that conform to the requirements of the data handler as well as to the requirements of the caller.

Figure 40 provides a visual overview of the data-handler 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 data-handler development, the major tasks of developing a data handler can be done in parallel by different members of the development team.

Figure 40. Overview of the data-handler development process

Tools for data-handler development

Because data handlers are written in Java, data handlers can be developed on either a Windows or UNIX system. Table 71 lists the tools that IBM provides for data handler development.

Table 71. Tools for data-handler development

Development Tool	Description
Adapter Development Kit (ADK)	Includes the following: • Sample data handlers • Stub file for extending `DataHandler` class
Data Handler API	Single class, `DataHandler`, which you extend to create a custom data handler.
Java Connector Development Kit (JCDK)	Contains Java classes to work with business objects.

The 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 the product directory.

Note: The ADK is part of the WebSphere Business Integration Adapters product and requires its own separate Installer. Therefore, to have access to the development samples in the ADK, you must the WebSphere Business Integration Adapters product and install the ADK. Please note that the ADK is availableonly for Windows systems.Table 72 lists the samples that the ADK provides for the development of a data handler as well as the subdirectory of the DevelopmentKits directory in which they reside.

Table 72. ADK samples for data handler development

Adapter Development Kit component	DevelopmentKits subdirectory
Data handler samples	`edk\DataHandler`

Sample data handlers

To assist with the development of a data handler, ADK includes code for several sample data handlers in the following product directory:

```
DevelopmentKits\edk\DataHandler\Samples
```

Table 73 lists the sample data handlers that the ADK provides.

Table 73. Sample data handlers included with the EDK

Name	Description
`delimited.java`	Converts business objects to Delimited strings and Delimited strings to business objects.
`fixedwidth.java`	Converts business objects to FixedWidth strings and from FixedWidth strings to business objects.
`namevalue.java`	Converts business objects to NameValue strings and NameValue strings to business objects.

Note: While these samples are useful to examine, they do *not* provide examples of all the functionality supported in the `DataHandler` class.

Development files

The `DevelopmentKits\edk\DataHandler` directory provides several files that assist in the development of a custom data handler, including those listed in Table 74.

Table 74. Data-handler development files

Data-Handler development file	For more information
`StubDataHandler.java`	"Extending the data handler base class" on page 171
`makeDataHandler.bat` (Windows systems)	"Adding a data handler to the jar file" on page 186

Data Handler API

The Data Handler API provides a single Java class, called `DataHandler`. The abstract `DataHandler` base class facilitates the development of a custom data handler. This class contains the methods that populate a business object with values extracted from input data, and methods that serialize a business object into a string or a stream. The class also includes utility methods that a custom data handler can use. You derive a custom data handler from the `DataHandler` class.

For information on the methods in the `DataHandler` class, see Chapter 11, "Data Handler base class methods," on page 195.

Java Connector Development Kit

To work with business objects, a data handler must use methods from the classes in the Java Connector Development Kit (JCDK). As you develop your data handler, you may need to import additional JCDK classes, such as `CxCommon.CxObjectContainerInterface` or `CxCommon.CxObjectAttr`. For reference information on the JCDK methods, see the *Connector Development Guide for Java* in the WebSphere Business Integration Adapters documentation set.

Note: The JCDK is the low-level Java connector library. Documentation of its methods are contained in a separate part of the *Connector Development Guide for Java*.

Designing the data handler

Before you begin to write a custom data handler, it is recommended that you clearly understand:

- The data format of the files that the data handler will be converting
- The business object model

  In particular, you need to know how to:
  - extract values from a business object instance
  - build a business object instance and populate it with values from a file

Creating a metadata-driven data handler

For your custom data handler to be metadata-driven, it must dynamically specify information that identifies the data handler to use. For more information on metadata-driven data handlers, see "Metadata-driven data handler design" on page 19.

Using data-handler meta-objects

One of the design decisions you need to make is whether you data handler will use meta-objects to initialize its configuration.

Note: For more information about meta-objects, see "Configuring data handlers" on page 24.

Consider the following when deciding whether to use meta-objects:

- Meta-objects allow a data handler to be dynamically configured. This design strategy creates a more flexible data handler, one that can be configured based on the context in which it is called.

  To call a data handler that uses meta-objects, the caller passes the data handler's associated MIME type into the `createHandler()` method. When called with a MIME type, `createHandler()` initializes a newly instantiated data handler with the configuration information in the child meta-object.
- There is overhead associated with the accessing and searching of meta-objects. Your data handler might want to avoid meta-objects if its configuration information does not change (it can be hard-coded) or if it converts data that does not have an associated MIME type.

  To call a data handler that does *not* use meta-objects, the caller passes *only* the data handler's class name into the `createHandler()` method. When called with a class name, `createHandler()` instantiates a data handler of that class; it does not search for associated meta-objects.

If you design your custom data handler to use meta-objects, you need to create these meta-objects as part of your data-handler implementation. For more information, see "Creating data-handler meta-objects" on page 187.

Extending the data handler base class

To create a custom data handler, you extend the data-handler base class (`DataHandler`) to create your own *data-handler class*. The `DataHandler` class contains methods that perform the conversions (string-to-business-object and business-object-to-string), as well as utility methods to assist in development. The EDK contains stub code and makefiles for a custom data handler. The stub file contains Java code that defines an empty class listing all the methods that you must implement. You can use the stub file as a template to generate a custom data handler.

To create a data handler source file using the stub file:

1. Copy the `StubDataHandler.java` file and rename it so that its name matches the name of the data-handler class that it defines.

   The stub file resides in the `DevelopmentKits\edk\DataHandler` subdirectory in the product directory. It includes `import` statements that import the data handler package `com.crossworlds.DataHandlers`. It also imports some classes from the Java Connector Development Kit.
2. Change the `StubDataHandler` keyword to the name of the class that implements your custom data handler.

   For example, the following line extends the `DataHandler` class to create a custom data-handler class called `HtmlDataHandler`:

   ```
   public class HtmlDataHandler extends DataHandler
   ```

Implementing the methods

To develop a data handler, you implement the following methods of the `DataHandler` class:

- The abstract `DataHandler` methods (required)
- The public `DataHandler` methods (optional)

The methods of your custom data handler go in the Java source file of the `DataHandler` class that you created in "Extending the data handler base class" on page 171.

Note: If the caller re-uses a cached instance of the `DataHandler` class over multiple threads, you might need to make the class thread-safe. To determine whether this is required, see the *Connector Development Guide for Java* for more details on the threading model.

Implementing the abstract methods

The data-handler base class, `DataHandler`, provides the abstract methods in Table 75, which you *must* implement in the `DataHandler` class for your custom data handler.

Table 75. Abstract methods in the DataHandler class

Data conversion	Format of serialized data	DataHandler method
String-to-business-object conversion	Converts serialized data, accessed through a `Reader` object, to a business object	getBO() - abstract
Business-object-to-string conversion	Converts a business object to an `InputStream` object	getStreamFromBO()
	Converts a business object to a `String` object	getStringFromBO()
	Converts a business object to a byte array.	getByteArrayFromBO()

Note: The copy of the `StubDataHandler.java` file that extends the `DataHandler` class with your custom data handler contains declarations for the abstract methods you need to implement.

The following sections provide implementation information for each of the abstract `DataHandler` methods.

Implementing conversion to a business object

The abstract `getBO()` method performs the string-to-business-object conversion; that is, it populates a business object with data extracted from a Java `Reader` object. There are two versions of the `getBO()` method:

- `getBO(Reader` *`serializedData`*`, BusinessObjectInterface` *`theObj`*`, Object` *`config`*`)`

  Input arguments include a `Reader` object that contains the serialized data and a reference to the business object. The method populates the *`theBusObj`* business object with the *`serializedData`* data.
- `getBO(Reader` *`serializedData`*`, Object` *`config`*`)`

Input arguments include a `Reader` object that contains the serialized data. The method determines the name of the business object type (the business object definition) from the data, then creates and fills a business object instance of that type.

Note: To support the data handler when it is called in the context of a connector, your data-handler class (which extends `DataHandler`) must implement *both* versions of the `getBO()` method. To support the data handler when it is called *only* from an access client (IBM WebSphere InterChange Server integration broker only), you must implement only the second form of the `getBO()` method; the Server Access Interface uses only this second form of `getBO()`.

The `getBO()` method allows the caller to pass in an optional object containing configuration information (the *`config`* parameter). This information is in addition to the configuration information specified in the data-handler meta-object. As an example, the configuration object can point to a template file or to a URL that the data handler uses.

Note: When converting to a business object, `getBO()` must ensure that any attribute identified with a `cw_mo_`*`label`* tag in the application-specific information of the parent business object does *not* get a value. For more information on the `cw_mo_`*`label`* tag, see "Implementing conversion from a business object" on page 177.

The purpose of the abstract `getBO()` method is to populate a business object with the serialized data that the `Reader` object contains. The public versions of `getBO()` can then receive the serialized data in one of several supported forms, convert the data to a Reader object, and then call the abstract `getBO()` method to perform the actual string-to-business-object conversion. For more information on the public `getBO()` method, see "getBO() - public" on page 198.

Figure 41 shows a basic implementation of the second form of the `getBO()` method. The example illustrates the steps in the conversion of data from a `Reader` object that contains fixed-width data to a business object:

1. The `getBO()` method converts the data in the `Reader` object to a `String` object. It then calls the user-defined `getBOFromString()` method to create an instance of the business object.
2. The `getBOFromString()` method determines what type of business object to create based on the first fixed-width token in the `String`, and then it creates a business object instance of that type. It gets the verb from the second fixed-width token in the `String` and sets the verb in the business object. This method then calls the user-defined `parseAttributeList()` method to parse the remainder of the `String` and populate the business object with values.
3. The `parseAttributeList()` method parses the `String` and recursively populates the business object. When the method finds an attribute of an object type, it determines whether the object is single or multiple cardinality. It calls `getMultipleCard()` to recursively process the business objects in the array and `getSingleCard()` to process a single-cardinality child business object.

Tip: A data handler extracts data from business objects and populates business objects with data in the same way that a connector does. For example, in the following code sample, the `getBOFromString()` method calls `JavaConnectorUtil.createBusinessObject()` to create a business object instance and `BusinessObjectInterface.setVerb()`to set the verb. For

information on how to process business objects, see the *Connector Development Guide for Java*.

```
public BusinessObjectInterface getBO(Reader serializedData,
    Object config)
    throws Exception
{
    clear(config);
    BusinessObjectInterface resultObj = null;

    // Create a String object from the Reader, then use the string
    // method
    int conversionCheck;
    char[] temp = new char[2000];
    StringBuffer tempStringBuffer = new StringBuffer(1000);

    while ( (conversionCheck = serializedData.read(temp)) != -1 )
        tempStringBuffer.append(new String (temp, 0, conversionCheck));

    mBOString = new String(tempStringBuffer);
    mBOStringLength = mBOString.length();

    resultObj = getBOFromString(null);
    return resultObj;
}


// Gets business object name and verb and creates a bus obj instance
private BusinessObjectInterface getBOFromString(String pvBOType)
    throws Exception
{
    BusinessObjectInterface returnObj = null;
    String lvBOName = null;
    String lvVerb = null;

    lvBOName = this.getNextToken(mBONameSize, true);
    lvVerb = this.getNextToken(mBOVerbSize, true);

    if( (pvBOType != null) && (lvBOName.compareTo(pvBOType) != 0) ) {
        throw new Exception(...);
    }
    else
    {
        returnObj = JavaConnectorUtil.createBusinessObject(lvBOName);
    }

    returnObj.setVerb(lvVerb);

    parseAttributeList(returnObj);

    return returnObj;
}
```

Figure 41. Example getBO() method (Part 1 of 4)

```
// Parse String to populate the attributes in the business object
protected void parseAttributeList(BusinessObjectInterface pvBO)
    throws Exception
{
    if ( mEndOfBOString )
        throw new Exception(...);
    else if( pvBO == null )
        throw new Exception(...);

    int lvAttrNum = pvBO.getAttrCount();

    String lvAttrName = null;
    String lvAttrValue = null;
    int lvAttrMaxLength = 0;

    try {
        for (int lvAttrIndex = 0; lvAttrIndex < lvAttrNum;
                lvAttrIndex++)
        {
            CxObjectAttr lvAttrSpec = pvBO.getAttrDesc(lvAttrIndex);
            lvAttrName = lvAttrSpec.getName();

            // Determine if the attribute is a simple attribute or a
            // business object container.
            if (lvAttrSpec.isObjectType())
            {
                // Get the next token based on the BOCountSize
                lvAttrMaxLength = mBOCountSize;
                lvAttrValue =  this.getNextToken(mBOCountSize, true);
                String lvBOType = lvAttrSpec.getTypeName();
                Object lvAttrBOValue = null;

                if (lvAttrSpec.isMultipleCard())
                {
                    this.getMultipleCard(pvBO,lvAttrIndex,lvBOType,
                        lvAttrValue);
                }
                else {
                    this.getSingleCard(pvBO,lvAttrIndex,lvBOType,
                        lvAttrValue);
                }
            }
            else
            {
                // Get the next token based on the MaxLength of the attribute
                lvAttrMaxLength = lvAttrSpec.getMaxLength();
                if (lvAttrMaxLength > 0)
                    lvAttrValue =  this.getNextToken(lvAttrMaxLength, false);
                else
                    lvAttrValue = null;
```

Figure 41. Example getBO() method (Part 2 of 4)

```
                // For simple String attribute, set to null, set to
                // configured CxIgnore or CxBlank values, or set to the
                // attribute value
                if (lvAttrValue == null )
                    pvBO.setAttrValue(lvAttrIndex, null);
                else if (lvAttrValue.equals(mCxIgnore)||
                      lvAttrValue.equals(""))
                    pvBO.setAttrValue(lvAttrIndex, null);
                else if (lvAttrValue.equals(mCxBlank)||
                      lvAttrValue.equals(" "))
                    pvBO.setAttrValue(lvAttrIndex, "");
                else
                    pvBO.setAttrValue(lvAttrIndex, lvAttrValue);
            }
        }
    }
}


// Populates a child container with values in the String
protected void getMultipleCard(BusinessObjectInterface pvParentBO,
    int pvParentAttrIndex, String pvBOType, String pvObjectCountString)
    throws CW_BOFormatException, Exception
{
    try {
        if ( pvObjectCountString.equals(mCxIgnore) )
        {
            // trace message
        }
        else {
            int lvObjectCount = Integer.parseInt(pvObjectCountString);
            if ( lvObjectCount == 0)
            {
                // trace message with the number of objects in container
            }
            else if (lvObjectCount > 0)
            {
                // There is at least one instance of the object in the string
                BusinessObjectInterface lvChildBO = null;

                // For each instance of the child object, parse the attribute
                // list, and then add the object container to the parent.
                for (int lvObjectIndex = 0; lvObjectIndex < lvObjectCount;
                      lvObjectIndex++)
                {
                    lvChildBO = getBOFromString(pvBOType);
                    pvParentBO.setAttrValue(pvParentAttrIndex,lvChildBO);
                }
            }
        }
    }
}
```

Figure 41. Example getBO() method (Part 3 of 4)

```
// Populates a single cardinality child with values in the String
protected void  getSingleCard(BusinessObjectInterface pvParentBO,
    int pvParentAttrIndex, String pvBOType, String pvObjectCountString)
    throws CW_BOFormatException, Exception
{
    try {
        BusinessObjectInterface lvChildBO = null;

        // Check the object count token
        // If it does not match "1", assume that the child object should
        //  be null
        if (pvObjectCountString.equals("1"))
        {
            // The string contains a single instance of the child
            lvChildBO = getBOFromString(pvBOType);
            pvParentBO.setAttrValue(pvParentAttrIndex, lvChildBO);
        }
        else if ( pvObjectCountString.equals(mCxIgnore) ||
              pvObjectCountString.equals("0"))
        {
            // Validate that the object count token is valid
        }
        else
            throw new CW_BOFormatException(...);
    }
}
```

Figure 41. Example getBO() method (Part 4 of 4)

Implementing conversion from a business object

The abstract methods in Table 76 perform the business-object-to-string conversion; that is, they each create serialized data in a particular format from a business object.

Table 76. Abstract methods to implement business-object-to-string conversion

Abstract method	Description
getStringFromBO()	Converts the data in a business object to a `String` object.
getStreamFromBO()	Converts the data in a business object to an `InputStream` object.
getByteArrayFromBO()	Cconverts the data in a business object to a byte array.

The goal of converting from a business object is to create a serialized form of all data in the business object. Sometimes, however, some business-object data should *not* be included in the serialized data. For example, a business object might use a child meta-object to hold dynamic configuration information for a connector.

IBM reserves all application-specific information that begins with the prefix `cw_mo_`*`label`* for configuration and/or dynamic metadata. To indicate any attribute that a data handler should *ignore* during conversion from a business object, the business object definition for the parent business object specifies the following tag in its application-specific information:

`cw_mo_`*`label=child_meta-object_attribute_name`*

where *label* is a string you define to further identify the purpose of the cw_mo_ tag and *child_meta-object_attribute_name* identifies the name of the attribute to ignore. This attribute usually contains the child meta-object. Multiple cw_mo_*label* tags are delimited by a semicolon (;).

When you implement the getStringFromBO(), getStreamFromBO(), and getByteArrayFromBO() methods for a custom data handler, these methods need to ensure that the data handler skips over connector-specific attributes, as follows:

1. Check for the existence of any cw_mo_*label* tag (where *label* is a string you provide to identify the attribute to ignore) in the application-specific information of the business object definition for the business object being converted.
2. If a cw_mo_*label* tag exists, locate the string that this tag provides (*child_meta-object_attribute_name*). Ignore any white space surrounding the equal sign (=).
3. While looping through the attributes of the business object, compare each attribute name against the *child_meta-object_attribute_name* string. Skip over any attribute with this name.

The following code fragment shows how to skip over attributes:

```
List configObjList =
     com.crossworlds.DataHandlers.text.namevalue.listMOAttrNames(BusObj);

//this list contains attribute names, which are configuration objects
for ( attributes in BusObj )
{
          String attrName = BusObj.getAttrDisc(i).getName();
          if ( configObjList.contains(attrName) )
          {
            //skip
            continue;
          }
}
```

For example, suppose a business object called MyCustomer uses a child meta-object to hold connector-specific routing information. If this meta-object is represented by an attribute named CustConfig, then MyCustomer could have the following tag in its application-specific information:

```
cw_mo_cfg=CustConfig
```

During conversion from a business object, a custom data handler checks the application-specific information for the business object definition associated with MyCustomer, locates the cw_mo_cfg tag, and determines that the CustConfig attribute needs to be skipped over. The resulting serialized data from the data handler does *not* include the CustConfig child meta-object.

Note: When converting from a business object, IBM-delivered data handlers skip over *any* attributes that the cw_mo_*label* tag identifies.

You need to develop any custom data handler to handle cw_mo_*label* tags *only* if the data handler works with connectors that use child meta-objects or other dynamic objects.

An implementation of the getStringFromBO() method: The getStringFromBO() method performs the business-object-to-string conversion; that is, it converts the data in a business object to a String object. For this method, the caller passes in

the business object to be converted. Figure 42 shows the `getStringFromBO()` method as implemented by the FixedWidth data handler. The method creates a `String` of fixed-width fields.

The example illustrates the steps in the conversion of data from a business object to a `Reader` object:

1. The `getStringFromBO()` method calls `setAttrList()` to recursively loop through the attributes in the business object. When the `setAttrList()` method finds a simple attribute, it calls the `setSimpleToken()` method to set the value.
2. The `setSimpleToken()` method adds the attribute value to a `StringBuffer` object, converts the `StringBuffer` to a `String` object, and adds the string to the `StringBuffer` representing the entire business object. When `setAttrList()` has processed the business object, the `getStringFromBO()` method converts the `StringBuffer` to a `String` object and returns the `String` to the caller.

```
public String getStringFromBO(BusinessObjectInterface theObj,
    Object config)
    throws Exception
{
    traceWrite(
      "Entering getStringFromBO(BusinessObjectInterface, Object) "
      + " for object type " + theObj.getName(),
      JavaConnectorUtil.LEVEL4);

    clear(config);
    String lvBOString = null;
    setAttrList(theObj);
    lvBOString = mBOStringBuffer.toString();

    traceWrite(
      "Exiting getStringFromBO(BusinessObjectInterface, Object) "
       + " for object type " + theObj.getName(),
       JavaConnectorUtil.LEVEL4);

    return lvBOString;
}


protected void setAttrList(BusinessObjectInterface pvBO) throws Exception
{
    traceWrite(
      "Entering setAttrList(BusinessObjectInterface) for object "
      + pvBO.getName(), JavaConnectorUtil.LEVEL4);

    int lvAttrNum = pvBO.getAttrCount();

    String lvAttrName = null;
    String lvAttrValue = null;
    int lvAttrMaxLength = 0;

    // Add the business object name and verb to the fixed width format
    //  String
    this.setSimpleToken( mBONameSize, pvBO.getName());
    this.setSimpleToken( mBOVerbSize, pvBO.getVerb());

    try {
        List moAttrNames = listMOAttrNames( pvBO );
        int lvAttrCount = pvBO.getAttrCount();

        ATTRIBUTE_WALK: for (int lvAttrIndex = 0;
              lvAttrIndex < lvAttrCount; ++lvAttrIndex)
        {
            CxObjectAttr lvAttrSpec = pvBO.getAttrDesc(lvAttrIndex);
            lvAttrName = lvAttrSpec.getName();

            // Check if the current attribute is a simple (String) type
            //  or a contained object.
            if (lvAttrSpec.isObjectType())
            {
                //skip child objects designated as meta objects
                if( moAttrNames.contains( lvAttrName ) )
                {
                    continue ATTRIBUTE_WALK;
                }
```

Figure 42. Example getStringFromBO() method (Part 1 of 5)

```
if (lvAttrSpec.isMultipleCard())
{
    CxObjectContainerInterface lvAttrMultiCardBOValue =
  (CxObjectContainerInterface) pvBO.getAttrValue(lvAttrIndex);

    if (lvAttrMultiCardBOValue == null)
    {
        traceWrite(
    "setAttrList found empty multiple cardinality container "
    + lvAttrSpec.getTypeName(), JavaConnectorUtil.LEVEL5);

        // Add the count to the fixed width String
        this.setSimpleToken( mBOCountSize, "0");
    }
    else
    {
        int lvObjectCount =
          lvAttrMultiCardBOValue.getObjectCount();
        traceWrite(
      "setAttrList found multiple cardinality container "
      + lvAttrSpec.getTypeName() + " with "
      + lvObjectCount + " instances",
      JavaConnectorUtil.LEVEL5);

        // Add the count to the fixed width String
        this.setSimpleToken( mBOCountSize,
          Integer.toString(lvObjectCount));

        // Add each object in the container to the fixed
        //  width String.
        for (int lvContObjectIndex = 0;
              lvContObjectIndex < lvObjectCount;
              ++lvContObjectIndex)
            setAttrList(
        lvAttrMultiCardBOValue.getBusinessObject(
            lvContObjectIndex));
    }
}
else
{
    BusinessObjectInterface lvAttrSingleCardBOValue =
  (BusinessObjectInterface) pvBO.getAttrValue(lvAttrIndex);

    if (lvAttrSingleCardBOValue == null)
    {
         traceWrite(
    "setAttrList found empty single cardinality container "
    + lvAttrSpec.getTypeName(), JavaConnectorUtil.LEVEL5);

         // Add the count to the fixed width String
         this.setSimpleToken( mBOCountSize, "0");
    }
```

Figure 42. Example getStringFromBO() method (Part 2 of 5)

```
                else
                {
                    traceWrite(
                  "setAttrList found single cardinality container "
                  + lvAttrSpec.getTypeName(),
                  JavaConnectorUtil.LEVEL5);

                    // Add the count to the fixed width String
                    this.setSimpleToken( mBOCountSize, "1");
                    setAttrList(lvAttrSingleCardBOValue);
                }
            }
        }
        else
        {
            lvAttrValue = (String) pvBO.getAttrValue(lvAttrIndex);
            lvAttrMaxLength = lvAttrSpec.getMaxLength();

            if (lvAttrMaxLength > 0)
                this.setSimpleToken(lvAttrMaxLength, lvAttrValue);
           }
        }
    }
    catch (CxObjectNoSuchAttributeException e)
    {
        throw new Exception(e.getMessage());
    }

    traceWrite(
        "Exiting setAttrList(BusinessObjectInterface) for object "
        + pvBO.getName(), JavaConnectorUtil.LEVEL4);
}


protected void setSimpleToken( int pvCellSize, String pvTokenValue)
    throws Exception
{
    traceWrite( "Entering setSimpleToken(int, String)",
      JavaConnectorUtil.LEVEL4);

    StringBuffer lvNewBuffer = new StringBuffer(pvCellSize);
    int lvTokenLength = 0;
    int lvCxIgnoreLength = mCxIgnore.length();
    int lvCxBlankLength = mCxBlank.length();

    int lvPadNumber = 0;


    // Check the token value to see if it is null
    if (pvTokenValue == null)
    {
        // For this case, we add the configured CxIgnore value to the
        //  fixed width String if it fits in the cell.
        if (!mTruncation && lvCxIgnoreLength > pvCellSize)
            throw new Exception(
                " Null attribute value encountered where cell size is "
                + pvCellSize + ", size of CxIgnore value is "
                + lvCxIgnoreLength + "and trucation is not allowed. "
                + "Please check your MO format configuration.");
```

Figure 42. Example getStringFromBO() method (Part 3 of 5)

```
    else
    {
        lvPadNumber = pvCellSize - lvCxIgnoreLength;
        lvNewBuffer.append(mCxIgnore);
    }

}
else if (pvTokenValue.equals(""))
{
    // For this case, the configured CxBlank value is added to the
    //  fixed width String if it fits in the cell.
    if (! mTruncation && lvCxBlankLength > pvCellSize)
        throw new Exception(
          " Blank attribute value encountered where cell size is "
          + pvCellSize + ", size of CxBlank value is "
          + lvCxBlankLength + "and trucation is not allowed. "
          + "Please check your MO format configuration.");
    else
    {
        lvPadNumber = pvCellSize - lvCxBlankLength;
        lvNewBuffer.append(mCxBlank);
    }
}
else
{
    // For this case, actually add the token value to the fixed
    // width String, unless the data is too long for the cell.
    lvTokenLength = pvTokenValue.length();
    if (!mTruncation && lvTokenLength > pvCellSize )
        throw new Exception(
          " Attribute value encountered where cell size is "
          + pvCellSize + ", size of token value is "
          + lvTokenLength + "and trucation is not allowed. "
          + "Please check your MO format configuration.");
    else
    {
        lvNewBuffer.append(pvTokenValue);
        lvPadNumber = pvCellSize - lvTokenLength;
    }
}

if (lvPadNumber <= 0 && mTruncation)
    // Token is longer than the cell and truncation is allowed,
    //  so the characters up to pvCellSize are added
    lvNewBuffer.setLength(pvCellSize);
else if (lvPadNumber > 0)
{
    // Pad the cell based on the configuration option chosen
    if ( mAlignment.equals(fixedwidth.AlignmentLeft) ||
          mAlignment.equals(fixedwidth.AlignmentBoth))
        this.padRight(lvNewBuffer, lvPadNumber);
    else if (mAlignment.equals(fixedwidth.AlignmentRight))
        this.padLeft(lvNewBuffer, lvPadNumber);
}
```

Figure 42. Example getStringFromBO() method (Part 4 of 5)

```
    String lvNewBuffString =  lvNewBuffer.toString();

    // Note that this may cause a performance issue when the tracing
    //  level is low, but in most cases it should not as any one token
    //  is *usually* not very long.
    traceWrite(
      "Adding the following token to the fixed width String: "
      + lvNewBuffString, JavaConnectorUtil.LEVEL5);

    // After the cell has been fully formatted, append to fixed width
    //  String being built
    mBOStringBuffer.append(lvNewBuffString);

    traceWrite( "Exiting setSimpleToken(int, String)",
      JavaConnectorUtil.LEVEL4);
}
```

Figure 42. Example getStringFromBO() method (Part 5 of 5)

An implementation of the getStreamFromBO() method: The `getStreamFromBO()` method converts the data in a business object to an `InputStream` object. Figure 43 shows an example implementation of the `getStreamFromBO()` method. In this implementation, `getStreamFromBO()` calls `getStringFromBO()` to build a `String` object containing the business object data, and then it converts the `String` to an `InputStream`. The method returns an `InputStream` object representing the data in the business object.

```
public InputStream getStreamFromBO(BusinessObjectInterface theObj,
    Object config)
    throws Exception
{
    clear(config);

    String BOstring;
    BOstring = getStringFromBO(theObj, config);

    return new ByteArrayInputStream( BOstring.getBytes() );
}
```

Figure 43. Example getStreamFromBO() method

Overriding public methods

In addition to the abstract `DataHandler` methods (which you *must* implement), you might also need to override some public methods of the `DataHandler` class (see Table 77) so they work optimally with your custom data handler.

Table 77. Public methods of the DataHandler class

Public DataHandler method	Description
getBO() - public	Converts serialized data(in one of several formats) to a business object.
getBOName()	Obtains the name of the business object from the serialized data.
getBooleanOption()	Gets a value of a Boolean configuration option from the data handler.
getOption()	Gets the value of a configuration option from the data handler.
setOption()	Sets a configuration option in the data handler.

Table 77. Public methods of the DataHandler class (continued)

Public DataHandler method	Description
traceWrite()	Calls a trace-write method for the appropriate context of the data handler: connector or the Server Access Interface (if your integration broker is InterChange Server).

Building a custom name handler

A data handler can call a *name handler* to extract the name of a business object definition from the serialized data. This task is needed during string-to-business-object conversion when the caller of the data handler does *not* pass in a business object to be populated with the serialized data. In this case, the data handler must create the business object before it can populate it. To create the business object, the data handler must know the name of the associated business object definition. It is the name handler that obtains this business object name.

Note: Currently, the XML and EDI data handlers use name handlers to obtain the name of the business object to create.

The task of creating and implementing a custom name handler includes the following general steps:

1. Create a class that extends the `NameHandler` class.
2. Implement the abstract `getBOName()` method that reads the serialized data and returns the business object name.
3. Compile the class and add it to the `DataHandlers\CustDataHandler.jar` file. For more information, see "Adding a data handler to the jar file" on page 186.
4. Set the default value of the `NameHandlerClass` attribute in the meta-object for the data handler.

Extending the NameHandler class

To create a custom name handler, you extend the name-handler base class (`NameHandler`) to create your own *name-handler class*. The `NameHandler` class contains the method to extract the name of a business object from serialized data. The package for this name-handler base class is `com.crossworlds.DataHandlers.NameHandler`.

To dervide a name-handler class, follow these steps:

1. Create a name-handler class that extends the NameHandler class.
2. Ensure that the name-handler class file imports the classes of the NameHandler package:

   ```
   import
   ```

3. Implement the `getBOName()` method, which is the abstract method in the `NameHandler` class. For more information, see "Implementing the getBOName() method" on page 186.

The definition of the `NameHandler` class follows:

```
// Imports
import java.lang.String;
import java.io.Reader;
import com.crossworlds.DataHandlers.Exceptions.MalformedDataException;

public abstract class NameHandler {
```

```
// Constructors
public NameHandler() { }

// Methods
public abstract String getBOName(Reader serializedData,
String boPrefix)
throws MalformedDataException;
}

/* This method was introduced so that the NameHandler would have
 * a reference to the DataHandler. The DataHandler base calls this
 * method after it instantiated the NameHandler:
 * eg. nh = (NameHandler)Class.forName(className).newInstance();
 *     nh.setDataHandler(this);
*/
public final void setDataHandler( DataHandler dataHandler )
{
dh = dataHandler;
}
```

To create your own name handler, extend the NameHandler abstract base class.

Implementing the getBOName() method

Extending the NameHandler class requires implementing the getBOName() method, which reads serialized data and returns the name of the business object associated with the data. The syntax for this method is:

```
public abstract String getBOName(Reader serializedData, String BOPrefix)
throws MalformedDataException
```

where:

serializedData
: Is a reference to an object that contains the message.

BOPrefix
: Is a String value that contains the business object prefix for the names of the business object definitions; this argument can be set to the BOPrefix attribute in the meta-object (if one exists).

Setting the meta-object attribute

To tell the data handler to use a custom name handler, you must set the Default Value property of a meta-object attribute to the full class name. The data handler can then obtain the class name at runtime from one of its configuration options. By default, this meta-object attribute is called NameHandlerClass. The child meta-object associated with both the XML and EDI data handlers include this attribute. The IBM-delivered default value for this attribute specifies the name of the default name handler class. To have a data handler use a custom name handler, make sure that you update the Default Value property for the NameHandlerClass attribute in the child meta-object associated with the data handler you extend.

Adding a data handler to the jar file

When you have completed the code for the new data handler, you must compile the class and add it to a Java archive (jar) file. The file CustDataHandler.jar is provided to contain custom data handlers. This jar file is located in the DataHandlers subdirectory of the product directory. To locate a data-handler class, the createHandler() method searches this jar file after it searches the CwDataHandler.jar file that contains delivered data handlers.

Note: To be able to compile Java code, you must have the Java Development Kit (JDK) installed on your machine. For the required version of the JDK and how to install it, refer to your product installation information.

To add a custom data handler to `CustDataHandler.jar`:

1. Edit the data-handler compilation script to add the names of the Java source files.

   This data-handler compilation script resides in the following subdirectory of the product directory:

   `DevelopmentKits\edk\DataHandler`

   Windows

   On a Windows system, the data-handler compilation script is called `make_datahandler.bat`. Add the names of the Java source files to the line:

   `set SOURCE_FILES_DH=`

   UNIX

   On a UNIX system, the data-handler compilation script is called `make_datahandler`. Add the names of the Java source files to the line:

   `SOURCE_FILES_DH=`

2. Run the data handler compilation script to compile the Java files into a `.class` file.
3. Add the new class to the `CustDataHandler.jar` file using the following command:

   `jar -vf CustDataHandler.jar` *`input_files`*

   where *`input_files`* is a list of the class files to add.

Creating data-handler meta-objects

If you write a custom data handler that uses data-handler meta-objects, you must:

- Create a child data-handler meta-object that contains attributes for the configuration information of the custom data handler.
- Modify the top-level data-handler meta-object to include the supported MIME type so that your data handler can be configured when it is instantiated.
- Set up the meta-objects in the business integration system.

Note: For information on how to determine whether to use meta-objects in the data-handler design, see "Using data-handler meta-objects" on page 171.

Creating the child meta-object

The child meta-object contains the configuration information for the data handler. The `createHandler()` method uses this information to initialize the newly instantiated data handler. For more detailed information on this process, see "Using a MIME type" on page 13.

To create a child meta-object for a custom data handler:

1. Create a child meta-object to represent an instance of the data handler.

You can use the Business Object Designer tool to create this child meta-object. The meta-object must contain attributes to define the configuration information that your data handler requires. At a minimum, the child meta-object *must* have a `ClassName` attribute.

2. Determine whether you need to specify the name of the data handler class in the `ClassName` attribute.
   - For a data handler that is in the default format, you do not need to specify a value for `ClassName`. The default format is:

     `com.crossworlds.DataHandlers.`*`MimeTypeString`*

     However, you can specify the class name for the data handler in the `ClassName` attribute's default value.
   - For a data handler class that is not in the default format, you must specify the full class name for the data handler instance. Otherwise, the `createHandler()` method cannot locate your data handler class when it tries to instantiate the data handler.
3. Set the default values of the appropriate attributes in the child meta-object to configure how the data handler instance will process data.

Modifying the top-level meta-object

When a caller supplies a MIME type to the `createHandler()` method, `createHandler()` determines which data handler to instantiate with these steps:

1. Locate the name of the top-level meta-object that is associated with the data handler.
2. Look in this top-level meta-object for a MIME type that matches the data to convert.
3. If this MIME type exists, find the name of the associated child meta-object, which contains the configuration information.

For a more detailed explanation of this process, see "Using a MIME type" on page 13. For this process to be successful, `createHandler()` must be able to locate the MIME type associated with data. Therefore, you must edit the top-level data-handler meta-object to include an attribute for the MIME type of the data that your data handler converts. This attribute must include:

- A name that matches the MIME-type string for the data handler's associated MIME type

  The name of the MIME type can contain *only* alphanumeric characters and the underscore (_). No other special characters are valid. If your MIME type contains a period (.), replace it with an underscore.
- A single-cardinality business object
- A type that is the name of the child meta-object representing your data handler

As an example, Figure 44 shows a top-level connector meta-object that is configured for a custom HTML data handler.

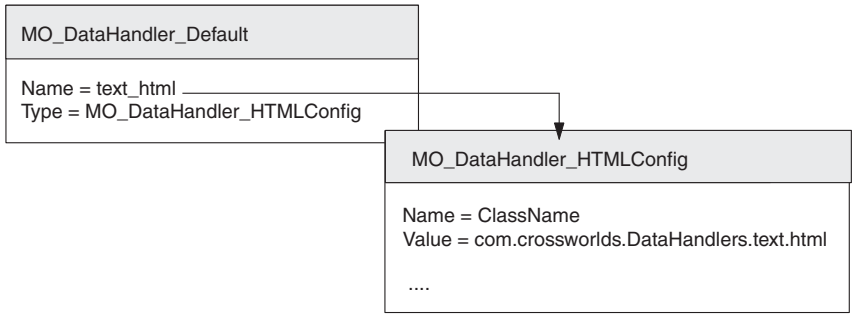

Figure 44. Example top-level connector meta-object for a custom data handler

In Figure 44, the default top-level meta-object for a connector (`MO_DataHandler_Default`) has been modified to support a new MIME type: HTML. In support of this MIME type, the `MO_DataHandler_Default` meta-object contains the following attribute properties:

```
Attribute Name = text_html
Attribute Type = MO_DataHandler_HTMLConfig
```

Important: The name of the MIME type is limited to alphanumeric characters and underscores (_). No other special characters can be used for the MIME type.

Setting up meta-objects in the business integration system

Once you have created the data-handler meta-objects, you must set up these meta-objects in the WebSphere business integration system, as follows:

1. Load the new meta-objects into the repository.
2. Modify the appropriate meta-object depending on the context in which the data handler will be called:
   - If the data handler is to execute in the context of a connector, add the data handler meta-object as a child to the top-level data handler meta-object. Then add support for the top-level data handler meta-object to the connector definition.
   - If your integration broker is InterChange Server and the data handler is to execute in the context of the Server Access Interface, add the data handler meta-object as a child to the top-level server meta-object `MO_Server_Datahandler`.

Setting up other business objects

In addition to coding the data handler, you must set up any business objects for the data handler. Create business objects that conform to the requirements of the data handler as well as to the requirements of the caller.

Tip: Make sure that any business objects the data handler requires are on the supported objects list for any connector that will use the data handler.

Configuring a connector

If the custom data handler will be used in the context of a connector, you need to configure each connector to obtain the name of the top-level connector meta-object. Connectors get information about data-handler meta-object names and class names in different ways. For example:

- To configure the WebSphere Business Integration Adapter for XML to use a data handler, you set the `DataHandlerConfigMO` connector configuration property and set the `MimeType` attribute in the XML connector's business object.
- To configure the WebSphere Business Integration Adapter for JText to use a data handler, you set the `ClassName`, or `DataHandlerConfigMO` and the `MimeType` attributes in the JText configuration meta-object.

For more information, see "Configuring a connector to use data handlers" on page 28.

Tip: When you configure a connector to use a data handler, make sure that the spelling of the meta-object name is correct and make sure that the MIME type is spelled correctly.

An internationalized data handler

An *internationalized data handler* is a data handler that has been written in such as way 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 data handler that handles locale-sensitive data is called the *internationalization* (I18N) of the data handler. Preparing an internationalized data handler for a particular locale is called the *localization* (L10N) of the data handler.

This section provides the following information on an internationalized data handler:

- What is a locale?
- Design considerations for an internationalized data handler

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 data handler

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

- Locale-sensitive design principles
- Character-encoding design principles

Locale-sensitive design principles

To be internationalized, a data handler 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 data handler should obtain its error messages from an English-language message file. The data handler framework that is installed with the product is internationalized. To complete the internationalization (I18N) of a data handler you develop, you must ensure that your data-handler implementation is internationalized.

An internationalized data handler must follow a set of locale-sensitive design principles:

- The text of all error, status, and trace messages should be isolated from the data-handler implementation in a message file and translated into the language of the locale.
- Sorting or collation of data uses a collation order appropriate for the language and country of the locale.
- String processing (such as comparison, substrings, and letter case) is appropriate for characters in the locale's language.
- Formats of dates, numbers, and times are appropriate for the locale.

The data handler might need to perform locale-sensitive processing (such as data format conversions) when it converts between the serialized data application and a business object. To track the locale associated with the data handler's environment, the `DataHandler` class has a private locale variable, which is initialized to the locale of the operating system on which the data handler runs. You can access the data handler environment's locale (the value of this private `locale` variable) at runtime through the accessor methods in Table 78.

Table 78. Methods to access the data handler environment's locale

Data Handler class	Method
`DataHandler`	getLocale(), setLocale()

When a business object is created, it has a locale associated with its data. 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`). To create a business object, your data handler can use the methods shown in Table 79. These methods have access to

the private locale variable in the DataHandler class. When one of these methods creates a business object, it associates with this business object the locale that the private DataHandler locale variable specifies.

Note: The methods in Table 79 set the locale *only* in the top-level business object. If the business object contains child business objects, these children might not have a valid locale value because they have their locale set to the system default.

Use the methods in Table 79 to create a business object and set the locale for its data. To ensure that the private locale variable specifies the correct locale for the data in the business object, you can use the setLocale() method *before* you call either of the methods in Table 79.

Table 79. Methods to assign a locale to a business object

Data Handler class	Method
DataHandler	getBO() - public, getBOName()

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 the Unicode character set. 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 IBM WebSphere business integration system 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.

Because a data handler is a component written in Java, it handles the serialized data in the Unicode code set. Usually, the source of the data's input stream is also processing in Unicode. Therefore, a data handler does not normally need to perform character conversion on the serialized data. However, if the input or output data contains a byte array whose character encoding is not the same as the system default, the data handler must provide the character encoding.

To track the character encoding associated with the data handler's environment, the DataHandler class has a private character-encoding variable, which is initialized to the character encoding associated with the locale of the operating system on which the data handler runs. You can access the data handler environment's character encoding (the value of this private character-encoding variable) at runtime through the accessor methods in Table 80.

Table 80. Methods to retrieve the data handler's character encoding

Data Handler Class	Method
`DataHandler`	getEncoding(), setEncoding()

Chapter 11. Data Handler base class methods

The DataHandler class is the base class for the data handlers. It is contained in the com.crossworlds.DataHandlers package. All data handlers, including custom data handlers, must extend this abstract class.

The methods documented in this chapter fall into three categories:

- Abstract methods that must be implemented
- Public methods that have a provided implementation that may be overridden if needed
- Static methods that are called by connectors or access clients

Table 81 lists the methods of the DataHandler class.

Table 81. Member methods of the DataHandler class

Member methods	Type	Description	Page
createHandler()	Public Static	Creates an instance of a data handler.	195
getBO() - abstract	Abstract	Populates a business object with values extracted from serialized input data.	196
getBO() - public	Public	Converts serialized data to a business object.	198
getBOName()	Public	Gets the name of a business object based on the content of the serialized data.	199
getBooleanOption()	Public	Gets the value of the specified data-handler configuration option if it contains Boolean data.	200
getByteArrayFromBO()	Public	Serializes a business object into a byte array.	201
getEncoding()	Public	Retrieves the character encoding that the data handler is using.	202
getLocale()	Public	Retrieves the locale of the data handler.	202
getOption()	Public	Gets the value of the specified data-handler configuration option (if it has been set).	203
getStreamFromBO()	Abstract	Serializes a business object into an InputStream object.	203
getStringFromBO()	Abstract	Serializes a business object into a String object.	204
setConfigMOName()	Static	Sets the name of the top-level data-handler meta-object in a static property of the DataHandler base class.	205
setEncoding()	Public	Sets the character encoding that the data handler is using.	206
setLocale()	Public	Sets the locale of the data handler.	206
setOption()	Public	Sets the value of the specified data-handler configuration option.	207
traceWrite()	Public	Calls the appropriate trace-write function to write a trace message for the data handler.	208

createHandler()

Creates an instance of a data handler.

Syntax

```
public static DataHandler createHandler(String className,
    String mimeType, String BOPrefix);
```

Parameters

className Is the class name of the data handler instance to create. If not specified, the method uses the *mimeType* argument to determine which data-handler class to instantiate.

mimeType Specifies the MIME type of the data handler instance to create. If not provided, the method expects a *className* value to be provided. Key to the meta-object. If *BOPrefix* is provided, *mimeType* becomes part of the key.

BOPrefix Is an optional parameter. If present, it is combined with *mimeType* to form the key to the meta-object. This argument can be used to specify a MIME subtype. It can also be used to set the data-handler configuration property `BOPrefix`.

Return values

An instance of a data handler.

Exceptions

`Exception`
Thrown if the method is unable to instantiate the data handler.

Notes

This method creates an instance of a data handler based on the values of its *className*, *mimeType*, and *BOPrefix* parameters:

- If the `createHandler()` method is called by a connector, the connector can specify a *className* value. If *className* is specified, `createHandler()` instantiates a data handler of this class name.
- If *mimeType* is specified, `createHandler()` creates a data handler based on the specified MIME type.

  The method checks the top-level data-handler meta-object for an attribute whose name matches the content type of either the *mimeType* parameter or the *mimeType* and *BOPrefix* combination. If a matching attribute is found, the value of the `ClassName` attribute in the child meta-object is used as the class name.

If the method succeeds in instantiating a class for the data handler, it calls `setupOptions()` to set up the configuration properties for use by the data handler instance. For a complete description of how `createHandler()` instantiates a data handler, see "Data handler instantiation" on page 12.

For example, for `MIME = "text/xml-application-xxx"`, the method loads the `com.crossworlds.DataHandlers.text.xml_application_xxx` class.

---

getBO() - abstract

Populates a business object with values extracted from serialized input data.

Syntax

```
public abstract void getBO(Reader serializedData,
        BusinessObjectInterface theBusObj, Object config);
public abstract BusinessObjectInterface getBO(Reader serializedData,
        Object config);
```

Parameters

serializedData	Is a Java `Reader` object that accesses the serialized data.
theBusObj	Is the business object to populate with data.
config	Is an optional object that contains additional configuration information for the data handler.

Return values

The first form of this method has no return value. The second form returns a business object.

Notes

This `getBO()` method is the abstract method that performs string-to-business-object conversion for a data handler; that is, it defines the way to convert generic serialized data (accessed with a `Reader` object) to a business object. This method has two forms:

- The first form populates the empty business object, *`theBusObj`*, passed in by the caller.
- The second form creates a business object instance and populates it.

Note: A data handler that is called in the context of the Server Access Interface needs to provide functionality *only* for the second form of the `getBO()` method.

Important: The `getBO()` method is an abstract method that has no default implementation.Therefore, the data handler class *must* implement this method.

You pass the serialized data into `getBO()` as a Java `Reader` object. However, because `Reader` is a base class, you actually pass an instance of one of several subclasses of the `Reader` class. Some of the `Reader` subclasses provide an implementation for the `mark()` operation and some do not. The `mark()` operation allows the caller to mark a particular position within the stream and then subsequently return to that position.

Note: To pass a `Reader` object into the `getBO()` method of the XML or EDI data handlers, you must ensure that the `Reader` subclass implements the `mark()` method. You can call the `isMarkSupported()` method of the `Reader` class to determine whether this method is support for the `Reader` object you are using. It is recommended that you pass in the serialized data as a `StringReader` object.

If you need to provide your data handler with more configuration information than is included in the meta-object, you can use the *`config`* option to pass in an object that contains this information. For example, *`config`* could be a template file or a string to a URL for a schema that is used to build an XML document from a business object.

If *`config`* is a business object type, you can implement the `getBO()` method to call `setupOptions(`*`config`*`)`. The `setupOptions()` method is defined in the `DataHandler` base class. This method uses the attribute names in the business object as property names and the default values as the values for those properties. It sets the values of the configuration properties in the object for use by the data handler.

Once you have implemented the abstract `getBO()` method, the component that calls the data handler can call one of the public string-to-business-object conversion methods, shown in Table 82.

Table 82. Public string-to-business-object conversion methods

Public string-to-business-object conversion method	Description
`getBO(Object serializedData, Object config)`	Convert serialized data in a generic `Object` to a business object
`getBO(String serializedData, Object config)`	Convert serialized data in a `Object` to a business object
`getBO(InputStream serializedData, Object config)`	Convert serialized data in an `InputStream` to a business object
`getBO(byte[] serializedData, Object config)`	Convert serialized data in a byte array to a business object

See also

getBO() - public

getBO() - public

Converts serialized data to a business object.

Syntax

```
public BusinessObjectInterface getBO(Object serializedData,
    Object config);
public BusinessObjectInterface getBO(String serializedData,
    Object config);
public BusinessObjectInterface getBO(InputStream serializedData,
    Object config);
public BusinessObjectInterface getBO(byte[] serializedData,
    Object config);
public void getBO(Object serializedData,
    BusinessObjectInterface theBusObj, Object config);
```

Parameters

serializedData Is a reference to the serialized data.

theBusObj Is the business object to populate with data.

config Is an optional object that contains additional configuration information for the data handler.

Return values

The first four forms return a business object populated with the data from the input `Object`, `String`, `InputStream`, or byte-array object. The fifth form populates the specified business object with the data from the serialized data.

Exceptions

`Exception`

Thrown if the method is unable to convert the serialized data to a business object.

NotImplementedException
Thrown if the public version of the getBO() method is not implemented.

Notes

This getBO() method is the public method to perform string-to-business-object conversion. The DataHandler base class includes the abstract forms of getBO() (as described on page 196), which must be implemented as part of the data-handler class. This public version of getBO() defines a set of utility methods that allow a component (such as a connector or access client) to specify the *serializedData* as an Object, String, or InputStream objects, or as a byte array. The method converts the specified serialized data to a Reader object and then calls one of the abstract getBO() methods to convert the Reader object to a business object.

The public getBO() method has the following forms:

- The first form calls the appropriate getBO() method based on the type of the *serializedData* object. For example, if the type of the data is String, the method calls getBO(String *serializedData*, Object *config*).
- The second, third and fourth forms create a Reader object from a String, InputStream, or byte-array object and call the abstract getBO() method to convert the Reader object to a business object.
- The fifth form is another utility that handles getBO() calls when the caller passes in a business object. It performs the following tasks:
  - determines the type of the serializedData object passed in
  - converts the object to a String or InputStream object, if appropriate
  - calls the abstract version, passing the business object as well as the data

For information on the *config* argument, see its description under the abstract form of getBO() (as described on page 196).

See also

getBO() - abstract

getBOName()

Gets the name of a business object based on the content of the serialized data.

Syntax

```
public String getBOName(Reader serializedData);
public String getBOName(String serializedData);
public String getBOName(InputStream serializedData);
```

Parameters

serializedData A reference to a Reader object containing the message.

Return values

Returns a String object containing the name of the business object. If no value exists for the NameHandlerClass attribute, this method returns null.

Exceptions

The second and third forms of getBOName() can throw the following exceptions:

MalformedDataException
Thrown if the serialized data (*serializedData*) is not in the correct format.

NotImplementedException
Thrown if the name handler is not implemented.

Notes

The `getBOName()` method creates an instance of a name handler to extract the name of the business object definition from the serialized data. It instantiates this name-handler object based on the value of the `NameHandlerClass` meta-object attribute. The name handler builds the business object name based on the contents of a message.

The `getBOName()` method has the following forms:

- The first form returns the business object name based on the `BOPrefix` meta-object attribute (if one exists) as well as on the return value from the `NameHandler` class.
- The second form creates a `Reader` object from the `String` and then calls the first form.
- The third form creates a `Reader` object from the `InputStream` and then calls first form.

Currently, only the following IBM-delivered data handler use this method:

- XML data handler

  The default name handler for the XML data handler calls the base class `getBOName(Reader data)`. If the data handler cannot handle the request, the `<!DOCTYPE Name` is used to extract the base name of the business object. The final name is formed as:

  ```
  BOPrefix + "_" + Name.getStreamFromBO()
  ```

- EDI data handler

  The default name handler for the EDI data handler obtains the name of the business object from the EDI name-handler lookup table.

To create your own name handler, extend the `NameHandler` abstract base class and override the `getBOName()` method.

For more information, see "Building a custom XML name handler" on page 86.

getBooleanOption()

Gets the value of the specified data-handler configuration option if it contains Boolean data.

Syntax

```
public boolean getBooleanOption(String name);
```

Parameters

name Name of the configuration option.

Return values

Returns the value of the `Boolean` type option.

Notes

The `createHandler()` method uses any child meta-object associated with the data handler to initialize its configuration options. You can use the `getBooleanOption()` method to obtain the value of one of these options, as long as the option contains a Boolean value.

---

getByteArrayFromBO()

Serializes a business object into a byte array.

Syntax

```
abstract byte[] getByteArrayFromBO(BusinessObjectInterface theBusObj,
      Object config);
```

Parameters

theBusObj Business object to be converted to a byte array.

config Optional object containing additional configuration information for the data handler.

Return values

A `byte` array containing serialized data that represents the specified business object.

Exceptions

`Exception`
Thrown if the method cannot convert the business object to a byte array of serialized data.

Notes

The `getByteArrayFromBO()` method performs business-object-to-byte conversion for a data handler. It converts the data in the *`theBusObj`* business object into a byte array (a Java `byte[]` object).

Important: The `getByteArrayFromBO()` method is an abstract method that has no default implementation.Therefore, the data handler class *must* implement this method.

If you need to provide your data handler with more configuration information than is included in the meta-object, you can use the *`config`* option to pass in an object that contains this information. For example, *`config`* could be a template file or a string to a URL for a schema that is used to build an XML document from a business object.

If *`config`* is a business object type, you can implement the `getByteArrayFromBO()` method to call `setupOptions(`*`config`*`)`. The `setupOptions()` method is defined in the `DataHandler` base class. This method uses the attribute names in the business object as property names and the default values as the values for those properties. It sets the values of the configuration properties in the object for use by the data handler.

See also

getBO() - public, getStreamFromBO(), getStringFromBO()

getEncoding()

Retrieves the character encoding that the data handler is using.

Syntax

```
public final String getEncoding();
```

Parameters

None.

Return values

A `String` containing the data handler's character encoding.

Notes

The `getEncoding()` method retrieves the data handler's character encoding. The character encoding is part of the locale, which defines cultural conventions for data according to language, country (or territory).This method is the accessor method that retrieves the value of a private character-encoding variable in the `DataHandler` class. This character encoding should indicate the character encoding of the serialized data that the data handler is processing.

This method is useful when the data handler needs to perform character-encoding processing, such as character conversion.

See also

setEncoding()

getLocale()

Retrieves the locale of the data handler.

Syntax

```
public final Locale getLocale();
```

Parameters

None.

Return values

A Java `Locale` object that describes the locale for the data handler's environment.

Notes

The `getLocale()` method retrieves the data handler's locale, which defines cultural conventions for data according to language, country (or territory), and a character encoding. This method is the accessor method that retrieves the value of a private locale variable in the `DataHandler` class. By default, the data handler's locale is the locale of the operating system on which the data handler runs.

This method is useful when the data handler needs to perform locale-sensitive processing.

See also

setLocale()

getOption()

Gets the value of the specified data-handler configuration option (if it has been set).

Syntax

```
public String getOption(String name);
```

Parameters

name Name of the configuration option.

Return values

A `String` object that contains the value of the option.

Notes

The `getOption()` method obtains the value of a configuration option. If the data handler has an associated child meta-object, the `createHandler()` method uses the default values of these meta-object attributes to initialize its configuration options. You can use the `getOption()` method to obtain the value of one of these options.

See also

setOption()

getStreamFromBO()

Serializes a business object into an `InputStream` object.

Syntax

```
abstract InputStream getStreamFromBO(BusinessObjectInterface theBusObj,
      Object config);
```

Parameters

theBusObj Business object to be converted to a stream.

config Optional object containing additional configuration information for the data handler.

Return values

An `InputStream` object containing serialized data representing a business object.

Exceptions

Exception
Thrown if the method cannot convert the business object to a stream of serialized data.

Notes

The `getStreamFromBO()` method performs business-object-to-stream conversion for a data handler. It converts the data in the *theBusObj* business object into a stream (a Java `InputStream` object).

Important: The `getStreamFromBO()` method is an abstract method that has no default implementation.Therefore, the data handler class *must* implement this method.

If you need to provide your data handler with more configuration information than is included in the meta-object, you can use the `config` option to pass in an object that contains this information. For example, *config* could be a template file or a string to a URL for a schema that is used to build an XML document from a business object.

If `config` is a business object type, you can implement the `getStreamFromBO()` method to call `setupOptions(`*config*`)`. The `setupOptions()` method is defined in the `DataHandler` base class. This method uses the attribute names in the business object as property names and the default values as the values for those properties. It sets the values of the configuration properties in the object for use by the data handler.

See also

getBO() - public, getByteArrayFromBO(), getStringFromBO()

---

getStringFromBO()

Serializes a business object into a `String` object.

Syntax

```
abstract String getStringFromBO(BusinessObjectInterface theBusObj,
      Object config);
```

Parameters

theBusObj	Business object to be converted to a `String`.
config	Optional object containing additional configuration information for the data handler.

Return values

A `String` object containing serialized data that represents the data in the business object.

Exceptions

Exception
Thrown if the method cannot convert the business object to a string of serialized data.

Notes

The `getStringFromBO()` method performs business-object-to-string conversion for a data handler. It converts the data in the *`theBusObj`* business object into a Java `String` object.

Important: The `getStringFromBO()` method is an abstract method that has no default implementation.Therefore, the data handler class *must* implement this method.

If you need to provide your data handler with more configuration information than is included in the meta-object, you can use the `config` option to pass in an object that contains this information. For example, *`config`* could be a template file or a string to a URL for a schema that is used to build an XML document from a business object.

If `config` is a business object type, you can implement the `getStreamFromBO()` method to call `setupOptions(`*`config`*`)`. The `setupOptions()` method is defined in the `DataHandler` base class. This method uses the attribute names in the business object as property names and the default values as the values for those properties. It sets the values of the configuration properties in the object for use by the data handler.

See also

getBO() - public, getByteArrayFromBO(), getStreamFromBO()

---

setConfigMOName()

Sets the name of the top-level data-handler meta-object in a static property of the `DataHandler` base class.

Syntax

```
public static void setConfigMOName(String name);
```

Parameters

name String containing the name of the data-handler meta-object.

Return values

None.

Exceptions

Exception
Thrown if the method set the specified top-level data-handler meta-object.

Notes

The top-level data-handler meta-object holds the supported MIME types and the names of their associated child meta-objects. For information on data-handler meta-objects, see "Configuring data handlers" on page 24.

setEncoding()

Sets the character encoding that the data handler is using.

Syntax

```
public final void setEncoding(String encodingName);
```

Parameters

encodingName
A `String` object containing the new value to assign as the data handler's character encoding.

Return values

None.

Notes

The `setEncoding()` method sets the data handler's character encoding. The character encoding is part of the locale, which defines cultural conventions for data according to language, country (or territory). This method is the accessor method that sets a private character-encoding variable in the `DataHandler` class. This character encoding should indicate the character encoding of the serialized data that the data handler is processing.

This method is useful when the data handler needs to perform character-encoding processing, such as character conversion.

See also

getEncoding()

setLocale()

Sets the locale of the data handler.

Syntax

```
public final void setLocale(Locale localeObject);
```

Parameters

localeObject
A Java `Locale` object containing the new locale to assign to the data handler's environment.

Return values

None.

Notes

The `setLocale()` method sets the data handler's locale, which defines cultural conventions for data according to language, country (or territory), and a character encoding. This method is the accessor method that sets a private locale variable in the `DataHandler` class. This locale should indicate the locale of the serialized data that the data handler receives and creates.

This method is useful when the data handler needs to change the data handler's locale. A possible use is to specify a different locale for the serialized data to be converted to a business object. Calling `setLocale()` before a call to `getBO()` changes the locale that `getBO()` uses when it associated a locale with the business object it creates.

See also

getLocale()

---

setOption()

Sets the value of the specified data-handler configuration option.

Syntax

```
public void setOption(String name, String value);
```

Parameters

name	Name of the configuration option.
value	Value of the configuration option.

Return values

None.

Notes

The `setOption()` method assigns a new value to the configuration option. If the data handler has an associated child meta-object, the `createHandler()` method uses the default values of these meta-object attributes to initialize its configuration options. If the data handler is called in the context of a connector, this child meta-object is in the connector-process memory. If your integration broker is InterChange Server and the data handler is called in the context of an access client, this meta-object is in the memory of the InterChange Server process. You can use the `setOption()` method to override the value of one of these options.

Note: Changing a configuration option with `setOption()` sets the value of the meta-object attribute in memory. It does not affect the value of the attribute in the repository.

See also

getOption()

traceWrite()

Calls the appropriate trace-write function to write a trace message for the data handler.

Syntax

```
public void traceWrite(String message, int level);
```

Parameters

message The message text to use for the trace message.

level An integer specifying the trace level for the message. The trace level is from 0-5, where 0 specifies no tracing and 5 specifies full tracing.

Return values

None.

Notes

The `traceWrite()` method is a wrapper method that calls the appropriate trace write function depending on the context in which the data handler runs. The default tracing is connector tracing. If your integration broker is InterChange Server and the data handler is run in the context of the Server Access Interface, the `traceWrite()` method sets the tracing subsystem before calling the trace-write method.

Appendix. Using the XML ODA

This chapter describes the XML ODA, an Object Discovery Agent (ODA) that generates business object definitions for XML documents. Because an XML document can have its schema defined by either a Document type definition (DTD) or a schema document, the XMLODA can use either of these data models to discover business object requirements specific to the XML document.

This chapter contains the following sections:

Installation and usage

This section discusses the following:

Installing the XML ODA

To install the XML ODA, use an IBM WebSphere Installer. For instructions on how to install this ODA with the Installer for IBM WebSphere Business Integration Adapters, see the *Implementing Adapters with WebSphere MQ Integrator Broker* or *Implementing Adapters with WebSphere Application Server*. For instructions on how to install this ODA with the InterChange Server Installer, see the *System Installation Guide for UNIX* or *for Windows*.

When the installation is complete, the following files are installed in the product directory on your system:

- `ODA\XML\XMLODA.jar`
- `ODA\messages\XMLODAAgent.txt`
- `ODA\messages\XMLODAAgent_ll_TT.txt files` (message files specific to a language (*ll*) and a country or territory (*TT*))
- `ODA\XML\start_XMLODA.bat` (Windows only)
- `ODA/XML/start_XMLODA.sh` (UNIX only)

Note: Except as otherwise noted, this document uses backslashes (\) as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes. All product pathnames are relative to the directory where product is installed on your system.

Before using the XML ODA

Before you run the XML ODA, verify that your system has the required files for the XML ODA. In particular, make sure that the ODA environment file has been installed in the `bin` subdirectory of your product directory.

UNIX

Make sure that the ODA environment file, `CWODAEnv.sh`, is installed in the following directory: *`ProductDir`*`/bin`.

Windows

Make sure that the ODA environment file, `CWODAEnv.bat`, is installed in the following directory: *`ProductDir`*`\bin`.

You must also make sure that the variables are correctly set in the startup script or batch file, which runs the ODA. Open for editing the shell (`start_XMLODA.sh`) or batch (`start_XMLODA.bat`) file and confirm that the values described in Table 83 are correct.

Table 83. Shell and batch file configuration variables

Variable	Explanation	Example
`set AGENTNAME`	Name of the ODA	`set AGENTNAME=XMLODA`
`set AGENT`	Name of the ODA's jar file	**UNIX:** `set AGENT = ${`*`ProductDir`*`}/ODA/XML/XMLODA.jar` **WINDOWS:** `set AGENT = %`*`ProductDir`*`%\ODA\XML\XMLODA.jar`
`set AGENTCLASS`	Name of the ODA's Java class	`set AGENTCLASS=com.crossworlds.oda.xml.XMLAgent`

After installing the XML ODA and setting configuration variables in the shell or batch file (see Table 83), you must do the following to generate business objects:

1. Launch the XML ODA.
2. Launch Business Object Designer.
3. Follow a six-step process in Business Object Wizard, a GUI interface that Business Object Designer provides to configure and run an ODA.

The following sections describe these steps in detail.

Launching the XML ODA

You can launch the XML ODA with the startup script appropriate for your operating system.

UNIX

```
start_XMLODA.sh
```

Windows

```
start_XMLODA.bat
```

Note: The Windows Installer provides shortcuts to startup the ODAs it installs. If you have used this Installer to install the XML ODA, you will find a shortcut to start it under the menu `Programs > IBM WebSphere Business Integration Adapters > Adapters > Object Discovery Agents`.

You configure and run the XML ODA using Business Object Designer. Business Object Wizard, which Business Object Designer starts, locates each ODA by the name specified in the `AGENTNAME` variable of each script or batch file. The default ODA name for this connector is `XMLODA`.

Running multiple instances of the XML ODA

You can run multiple instances of an XML ODA either on the local host machine or a remote host machine. Each instance runs on a unique port. You can specify this port number as part of launching the ODA from within Business Object Designer. Figure 45 on page 213 illustrates the window in Business Object Designer from which you select the ODA to run.

Working with error and trace message files

Error and trace message files (the default is `XMLODAAgent.txt`) are located in the `\ODA\messages`, subdirectory under the product directory. These files use the following naming convention:

AgentName`Agent.txt`

If you create multiple instances of the ODA script or batch file and provide a unique name for each represented ODA, you can have a message file for each ODA instance. Alternatively, you can have differently named ODAs use the same message file. There are two ways to specify a valid message file:

- If you change the name of an ODA and do not create a message file for it, you must change the name of the message file in Business Object Designer as part of ODA configuration. Business Object Designer provides a name for the message file but does not actually create the file. If the file displayed as part of ODA configuration does not exist, change the value to point to an existing file.
- You can copy the existing message file for a specific ODA, and modify it as required. Business Object Designer assumes you name each file according to the naming convention. For example, if the AGENTNAME variable specifies `XMLODA1`, the tool assumes that the name of the associated message file is `XMLODA1Agent.txt`. Therefore, when Business Object Designer provides the file name for verification as part of ODA configuration, the file name is based on the ODA name. Verify that the default message file is named correctly, and correct it as necessary.

Important: Failing to correctly specify the message file's name when you configure the ODA causes it to run without messages. For more information on specifying the message file name, see Table 85 on page 214.

During the configuration process, you specify:

- The name of the file into which the XML ODA writes error and trace information
- The level of tracing, which ranges from `0` to `5`.

Table 84 describes these values.

Table 84. Tracing levels

Trace Level	Description
0	Logs all errors
1	Traces all entering and exiting messages for method
2	Traces the ODA's properties and their values
3	Traces the names of all business objects
4	Traces details of all spawned threads
5	• Indicates the ODA initialization values for all of its properties • Traces a detailed status of each thread that the XML ODA spawned • Traces the business object definition dump

For information on where you configure these values, see Table 85 on page 214.

Using an XML ODA in Business Object Designer

This section describes how to use Business Object Designer to generate business object definitions using the XML ODA. For information on launching Business Object Designer, see the *Business Object Development Guide*. Business Object Designer provides a wizard, called Business Object Wizard, that guides you through each of these steps. After you launch an ODA, you must launch Business Object Designer to obtain access to Business Object Wizard (which configures and runs the ODA). There are six steps in Business Object Wizard to generate business object definitions using an ODA.

After starting the ODA, do the following to start the wizard:

1. Open Business Object Designer.
2. From the File menu, select the New Using ODA... submenu.

   Business Object Wizard displays the first window in the wizard, named Select Agent. Figure 45 on page 213 illustrates this window.

To select, configure, and run the ODA, follow these steps:

1. "Select the ODA"
2. "Specify configuration properties" on page 213
3. "Expand nodes and select XML elements" on page 215
4. "Confirm selection of objects" on page 216
5. "Generate the business object definition" on page 217 and, optionally, "Provide additional information" on page 218
6. "Save the business object definition" on page 220

Select the ODA

Figure 45 on page 213 illustrates the first dialog box in Business Object Wizard's six-step wizard. From this window, select the ODA to run.

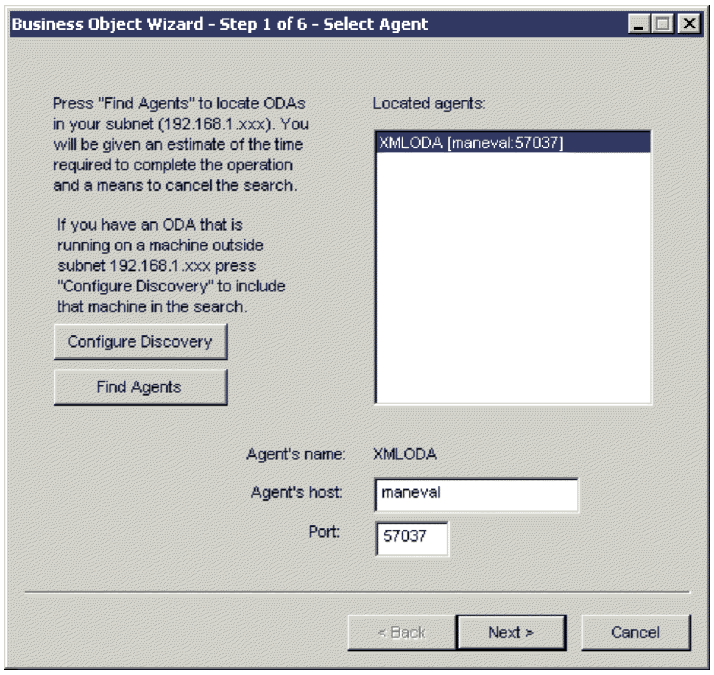

Figure 45. Selecting the ODA

To select the ODA:

1. Click the Find Agents button to display all registered or currently running ODAs in the Located agents field. Alternatively, you can find the ODA using its host name and port number.

   Note: If Business Object Wizard does not locate your desired ODA, check the setup of the ODA.

2. Select the desired ODA from the displayed list.

   Business Object Wizard displays your selection in the Agent's name field.

Specify configuration properties

The first time Business Object Wizard communicates with XML ODA, it prompts you to enter a set of ODA configuration properties as shown in Figure 46 on page 214.

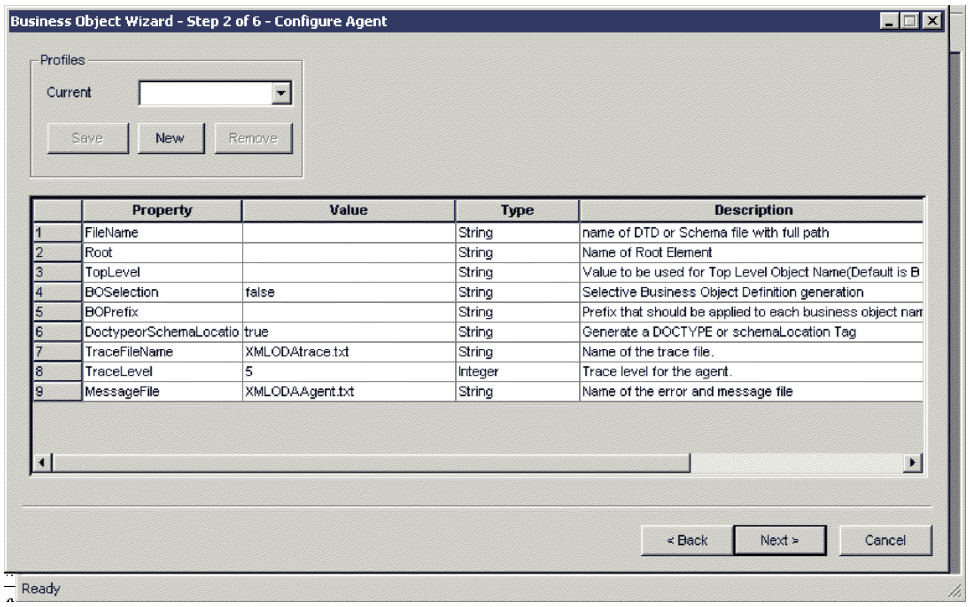

Figure 46. Specifyng ODA configuration properties

Configure the XML ODA properties described in Table 85.

Table 85. XML ODA configuration properties

Row number	Property name	Property type	Description
1	`FileName`	String	Full pathname of the DTD or schema document. A DTD file must have the `.dtd` extension; a schema-document file must have the `.xsd` extension.
2	`Root`	String	Name of the XML element that is to be treated as the root element. If no root element is specified, the XML ODA makes the following assumptions: • If the ODA is parsing a DTD, it treats the first XML element as the root. • If the ODA is parsing a schema document, it treats the first global element as the root.
3	`TopLevel`	String	Name to be used for the top-level business object that the ODA generates. The ODA prepends the top-level business object with the business-object prefix (which the `BOPrefix` property specifies), separated by an underscore (_). If you do *not* specify a top-level name, the ODA assigns the name *`BOPrefix_Root`* (where *BOPrefix* and *`Root`* are the values of the `BOPrefix` and Root properties) as the name of the top-level business object.
4	`BOSelection`	String	A boolean value (`true` or `false`) to indicate whether the XML ODA will allow you to select the names of elements for which business object definitions are to be used. • If this property is set to `false`, the XML ODA allows the user to select only the root as the element and will generate business object definitions for the root and all of its child elements. • If this property is set to `true`, the XML ODA allows the user to select any element and will generate business object definitions only for the selected elements. The default is `false`.

Table 85. XML ODA configuration properties (continued)

Row number	Property name	Property type	Description
5	BOPrefix	String	Prefix that the ODA applies to the name of each business object definition for the XML document. If you do not specify a business-object prefix, the ODA does *not* prepend any string to the name of the business object definition.
6	DoctypeorSchemaLocation	String	A boolean value (true or false) to indicate whether the XML ODA should generate attributes for: • When processing DTDs: the DOCTYPE tag • When processing schema documents: the schemaLocation and xsi attributes (in the XML Schema Instance namespace) The default is true.
7	TraceFileName	String	Full pathname of the file into which XML ODA writes trace information. If the file does not exist, XML ODA creates it in the specified directory. If the file already exists, XML ODA appends to it. By default, the XML ODA creates a trace file named XMLODAtrace.txt in the ODA\XML subdirectory of the product directory. Use this property to specify a different name for the trace file.
8	TraceLevel	Integer	Level of tracing enabled for XML ODA. Valid values are zero through five (0-5). Property defaults to a value of 5 (full tracing enabled). For more information, see "Working with error and trace message files" on page 211.
9	MessageFile	String	Full pathname of the error and message file. By default, the XML ODA creates a message and error file named XMLODAAgent.txt. **Important**: The error and message file *must* be located in the ODA\messages subdirectory of the product directory. Use this property to verify or specify an existing file.

Important: Correct the name of the message file if the default value displayed in Business Object Designer represents a non-existent file. If the name is not correct when you move forward from this dialog box, Business Object Designer displays an error message in the window from which the ODA was launched. This message does not pop up in Business Object Designer. Failing to specify a valid message file causes the ODA to run without messages.

You can save these properties in a named profile so that you do not need to re-enter them each time you use XML ODA. For information on specifying an ODA profile, see the *Business Object Development Guide*.

Expand nodes and select XML elements

Business Object Designer uses the properties configured in the previous step to connect the tool to the specified XML schema (DTD or schema document). After connecting, Business Object Designer displays a tree whose nodes represent all the XML elements defined in the XML schema.

You can expand the top-level XML element to display the entire hierarchical representation. For each XML element, XML ODA creates a child business object definition.

Figure 47 illustrates this dialog box with some XML elements expanded.

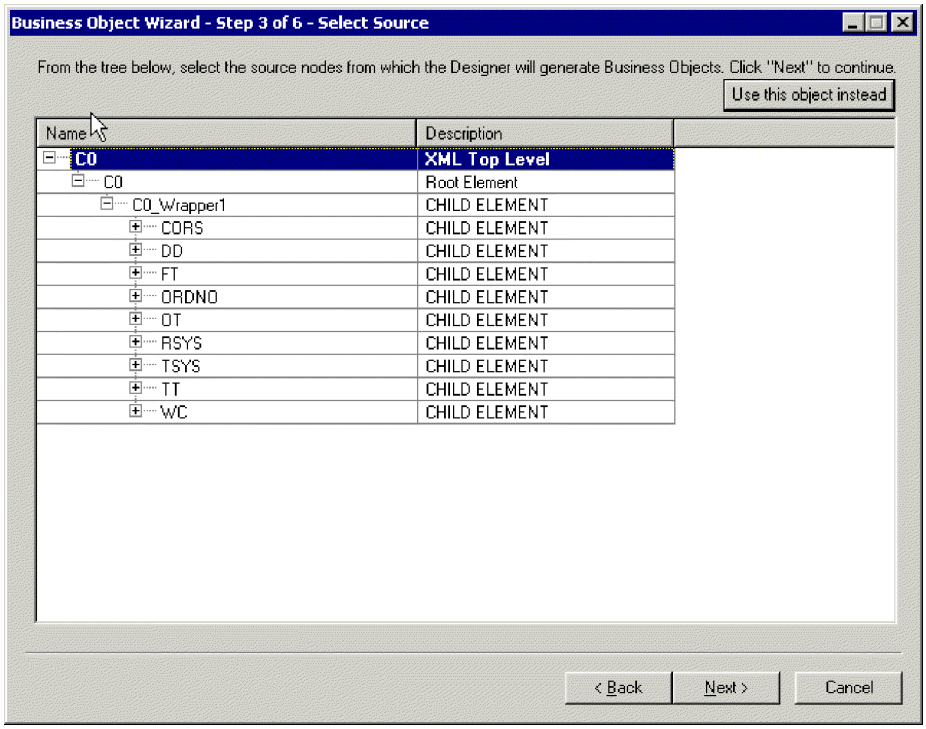

Figure 47. Tree of XML elements with expanded nodes

Select all required XML elements and click Next.

Confirm selection of objects

After you identify all the XML elements to be associated with the generated business object definitions, Business Object Designer displays the dialog box with only the selected objects. Figure 48 on page 217 illustrates this dialog box.

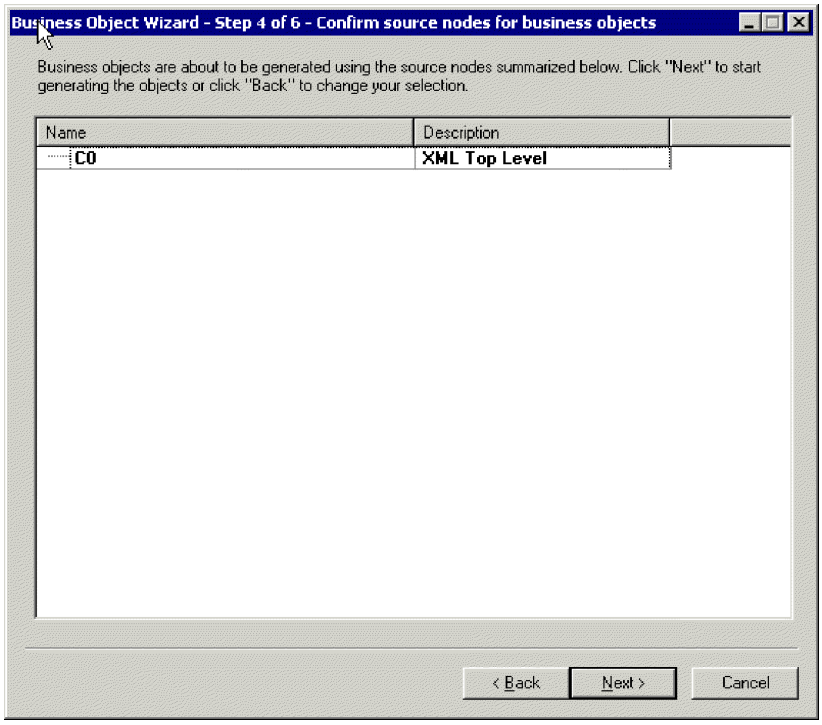

Figure 48. Confirming selection of objects

This window provides the following options:

- To confirm the selection, click Next.
- If the selection is not correct, click Back to return to the previous window and make the necessary changes. When the selection is correct, click Next.

Generate the business object definition

After you confirm the XML elements, the next dialog box informs you that Business Object Designer is generating the business object definition. If a large number of Component Interfaces has been selected, this generation step can take time.

Figure 49 on page 218 illustrates this dialog box.

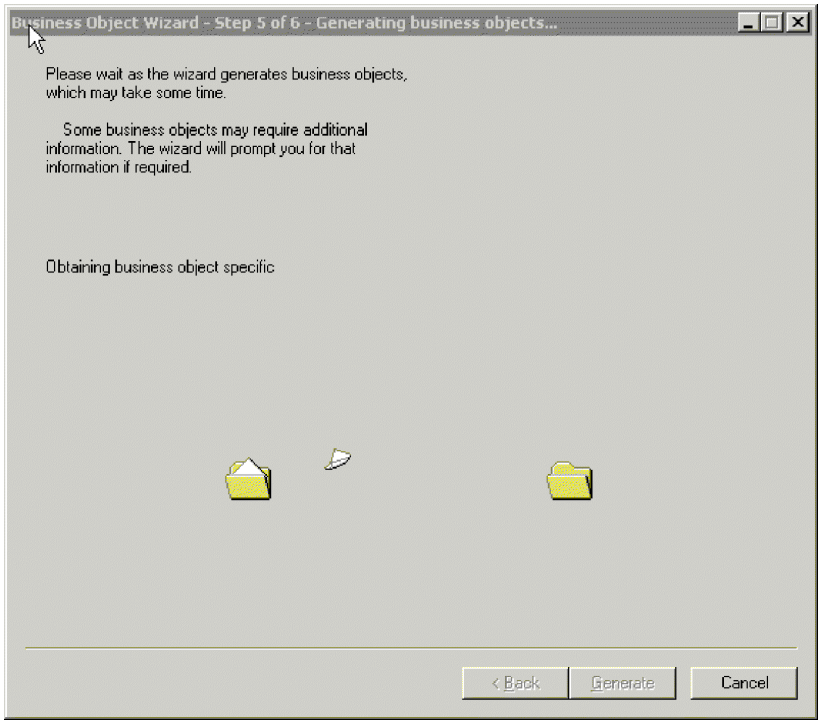

Figure 49. Generating the business object definitions

The XML ODA generates the name for a business object definition from the following information:

- The value of the `BOPrefix` ODA configuration property
- The value of the `TopLevel` ODA configuration property
- The name of the XML element that the business object definition represents

It separates each of these values with an underscore (_) character. Therefore, the name it generates has the following format:

`BOPrefix_TopLevel_XMLelement`

Provide additional information

Because the XML ODA needs additional information about the verbs, Business Object Designer displays the BO Properties window, which prompts you for the information. Figure 50 on page 219 illustrates this dialog box.

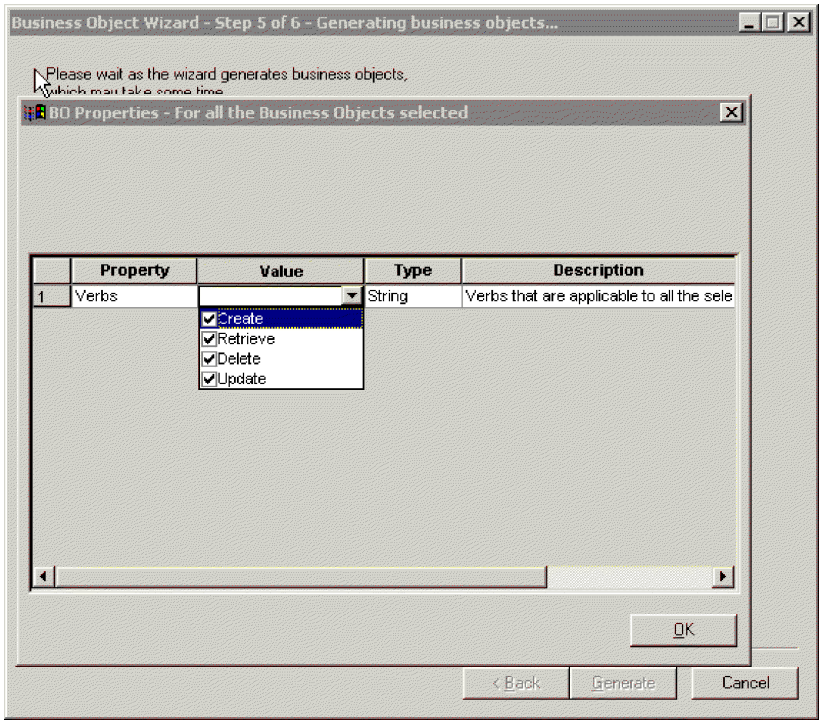

Figure 50. Providing additional information - verbs

In the BO Properties window, enter or change the verb information. Click in the *Value* field and select one or more verbs from the pop-up menu. These are the verbs supported by the business object.

Note: If a field in the BO Properties dialog box has multiple values, the field appears to be empty when the dialog box first displays. Click in the field to display a drop-down list of its values.

If your XML document has a schema document that contains an `anyAttribute` element, the XML ODA displays an additional BO Properties window, as shown in Figure 51.

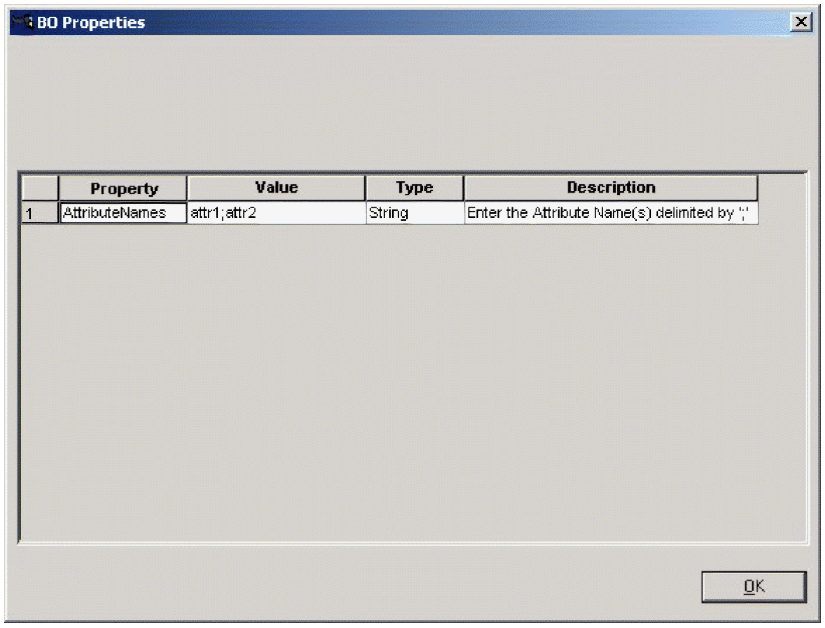

Figure 51. Providing additional information - attribute names

In this BO Properties window, enter the names of the business object attributes you want the XML ODA to create. Separate each attribute with a semicolon (;). For more information on `anyAttribute`, see "Supported schema-document structures" on page 78.

Save the business object definition

After you provide all required information in the BO Properties dialog box and click OK, Business Object Designer displays the final dialog box in the wizard. In this dialog box, you can take any of the following actions:

- Save the business object definition to the server (if InterChange Server is the integration broker).
- Save the business object definition to a file (for any integration broker).
- Open the business object definition for editing in Business Object Designer.

For more information, and to make further modifications, see the *Business Object Development Guide*.

Figure 52 on page 221 illustrates this dialog box.

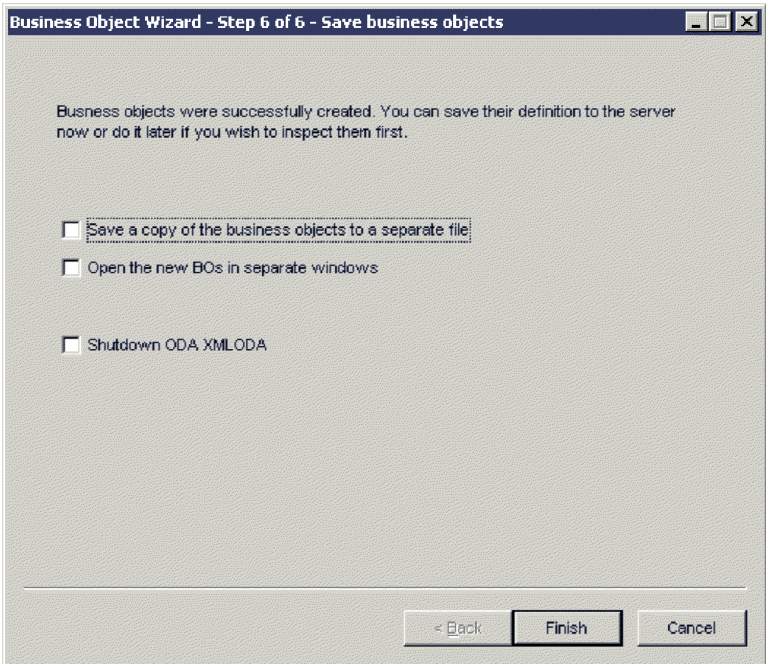

Figure 52. Saving the business object definition

Contents of the generated business object definition

The business object definition that the XML ODA generates the attributes, verbs, and application-specific information as described in the following sections:

Component of business object definition	For more information
Attributes	"Business object structure" on page 35 "Business object attribute properties" on page 36
Application-specific information	"Application-specific information" on page 38
Verbs	"Business object verbs" on page 39

Note: Earlier versions of the XML ODA generated parent business object definitions that designated the ObjectEventId attribute as the key. Business Object Designer no longer allows business object definitions to specify ObjectEventId as a key attribute. Therefore, XML ODA now takes special steps to avoid this behavior. This new behavior requires that you modify the generated business object definition to designate a key attribute. For more information, see "Key and Foreign Key attribute properties" on page 36.

Modifying information in the business object definition

It may be necessary modify information in the business object definition that the XML ODA creates. For example, you must manually remove unwanted attributes and add required tags for attribute application-specific information. To examine or modify the business object definition, you can use Business Object Designer or a text editor. To reload a revised definition into the repository, you can use Business Object Designer. Alternatively, if InterChange Server (ICS) is the integration broker, you can use the `repos_copy` command to load the definition into the repository; if

WebSphere MQ Integrator Broker is the integration broker, you can use a system command to copy the file into the repository directory.

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.

COPYRIGHT LICENSE

This information may contain sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

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. IBM WebSphere InterChange Server V4.2.2, IBM WebSphere Business Integration Toolset V4.2.2, IBM WebSphere Business Integration Collaborations V4.2, IBM WebSphere Business Integration Adapter Framework V2.4.0

Index

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

X

Printed in USA