

IBM WebSphere Business
Integration Server



Collaboration Development Guide

Version 4.3.0

WebSphere software

IBM WebSphere Business
Integration Server



Collaboration Development Guide

Version 4.3.0

Note!

Before using this information and the product it supports, read the information in "Notices" on page 475.

30September2004

This edition of this document applies to IBM WebSphere InterChange Server (5724-178) Version 4.3.0, IBM WebSphere Business Integration Toolset (5724-177) version 4.3.0, and all subsequent releases and modifications until otherwise indicated in new editions.

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

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

© Copyright International Business Machines Corporation 1999, 2004. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Preface	xiii
Audience	xiii
Scope of this manual	xiii
How to use this manual	xiv
Related documents	xv
Typographic conventions	xvi
Summary of Changes	xvii
New in release 4.3	xvii
New in release 4.2.2	xvii
New in release 4.2	xviii
New in release 4.1.1	xix
New in release 4.1.0	xix
New in release 4.0.1	xx
New in release 4.0.0	xx
Part 1. Getting started.	1
Chapter 1. Introduction to collaboration development	3
What are collaborations?	3
Tools for collaboration development	10
Overview of the development process	12
Chapter 2. Overview of Process Designer.	15
Starting Process Designer	15
Process Designer layout	16
Process Designer windows	17
Process Designer menus	20
Process Designer toolbars	23
Customizing the main window	23
Part 2. Creating a collaboration template	27
Chapter 3. Designing a collaboration.	29
CollaborationFoundation template	29
Extending CollaborationFoundation	35
WrapperFoundation template	47
Building collaboration groups	52
Including Web Services	53
Designing for long-lived business processes	54
Dynamic service calls	54
Designing for parallel execution	55
Examples	61
An internationalized collaboration	63
Designing collaborations for bidirectional scripts	69
Configuring database connection resilience feature	72
Handling failed events	75
Chapter 4. Building a collaboration template	81
Creating a collaboration template	81
Providing template property information	83
Defining scenarios	96
Creating the message file	100

Compiling a collaboration template	101
Converting templates.	102
Deleting a collaboration template.	107
Testing a collaboration	107
Debugging a collaboration	108
Chapter 5. Using activity diagrams	109
Using the diagram editor functionality	109
Activity diagram symbols	110
Action nodes	112
Transition Links	116
Decision nodes	120
Service calls	126
Subdiagrams	134
Iterators	139
Using other features of the Symbols toolbar	142
Obtaining values of collaboration configuration properties	142
Using transactional features	143
Terminating the execution path	143
Other activity diagram operations	145
Chapter 6. Using Activity Editor	149
Starting Activity Editor	149
The Activity Editor interface	149
Activity definitions	154
Supported function blocks	157
Example: Changing a date format	159
Example: Creating a duplicate business object	161
Importing Java packages and other custom code	162
Chapter 7. Handling exceptions	167
What is a collaboration exception?	167
How exceptions are processed.	168
How to handle exceptions	171
Handling particular service-call exceptions	178
Exceptions from the Collaboration API.	181
Chapter 8. Workspace and layout options	183
Aligning symbols	183
Nudging symbols	185
Zooming or panning on symbols.	186
Using the workspace grid	186
Changing display: user preferences	187
Hiding the Symbol Properties dialog boxes	191
Chapter 9. Coding tips and examples	193
Operations on the collaboration	193
Operations on business objects	203
Executing database queries.	212
Chapter 10. Creating a message file.	231
Operations that use the message file.	231
Creating a message file	231
Message file: Name and location	232
Explanations	233
Message parameters	233
Maintaining the file	234
Part 3. Supported function blocks	235

Chapter 11. Business object function blocks	237
Copy	238
Duplicate.	239
Equal Keys	239
Equals.	240
Exists	240
Get Boolean	241
Get Business Object	241
Get Business Object Array	242
Get Business Object Type	243
Get BusObj At	243
Get Double	243
Get Float	244
Get Int	245
Get Locale	245
Get Long	246
Get Long Text	246
Get Object	247
Get String	248
Get Verb	248
Is Blank	249
Is Business Object	249
Is Key	250
Is Null	250
Is Required	251
Iterate Children	251
Keys to String	251
New Business Object	252
New Business Object Array.	252
Set BusObj At	253
Set Content	253
Set Default Attribute Values	253
Set Keys	254
Set Locale	254
Set Value	254
Set Value By Position.	255
Set Value with Create.	256
Set Verb	256
Set Verb with Create	257
Shallow Equals	257
Size	258
To String	258
Valid Data	258
Verb:Create	259
Verb>Delete	259
Verb:Retrieve	260
Verb:Update.	260
Chapter 12. Business object array function blocks	261
Add Element	261
Duplicate.	262
Equals.	262
Get Element At.	263
Get Elements	263
Get Last Index	263
Is Business Object Array.	264
Max Attribute Value	264
Max Business Object Array	265
Max Business Objects.	266
Min Attribute Value	267
Min Business Object Array	267

Min Business Objects	268
Remove All Elements	269
Remove Element	269
Remove Element At	270
Set Element At	270
Size	271
Sum	271
Swap	272
To String	272

Chapter 13. Collaboration template function blocks 273

AnyException	274
AttributeException	274
Get Locale	274
Get Message	274
Get Message with Parameter	275
Get Name	275
Get Property	276
Get Property Array	276
Implicit DB Bracketing	276
Is Trace Enabled	277
JavaException	277
ObjectException	277
OperationException	278
Property Exists	278
Raise Collaboration Exception	278
Raise Collaboration Exception 1	279
Raise Collaboration Exception 2	280
Raise Collaboration Exception 3	280
Raise Collaboration Exception 4	281
Raise Collaboration Exception 5	282
Raise Collaboration Exception with Parameter	282
Send Email	283
ServiceCallException	284
SystemException	284
TransactionException	284

Chapter 14. Database connection function blocks. 285

Begin Transaction	285
Commit	286
Execute Prepared SQL	287
Execute Prepared SQL with Parameter	288
Execute SQL	288
Execute SQL with Parameter	289
Execute Stored Procedure	290
Get Database Connection	291
Get Database Connection with Transaction	291
Get Next Row	292
Get Update Count	293
Has More Rows	294
In Transaction	294
Is Active	295
Release	295
Roll Back	296

Chapter 15. Database stored procedure function blocks 299

Get Param Type	299
Get Param Value	300
New DB Stored Procedure Param	300

Chapter 16. Exception function blocks	303
Catch Collaboration Exception.	303
Get Message.	303
Get Message Number	304
Get Subtype.	304
Get Type	305
To String	306
Chapter 17. Execution function blocks	309
Get Context	309
MAPCONTEXT	309
New Execution Context	309
Set Context	310
Chapter 18. Date function blocks	311
Add Day	311
Add Month	311
Add Year	312
Date After	312
Date Before	312
Date Equals	313
Format Change.	313
Get Day	313
Get Month	313
Get Year	314
Get Year Month Day	314
Now	314
yyyy-MM-dd	314
yyyyMMdd	315
yyyyMMdd HH:mm:ss	315
Chapter 19. Logging and tracing function blocks	317
Log error	317
Log Error ID	317
Log Error ID 1	318
Log Error ID 2	318
Log Error ID 3	318
Log Information	319
Log Information ID	319
Log Information ID 1.	319
Log Information ID 2.	319
Log Information ID 3.	320
Log Warning	320
Log Warning ID	320
Log Warning ID 1.	320
Log Warning ID 2.	321
Log Warning ID 3.	321
Trace	321
Trace ID 1	321
Trace ID 2	322
Trace ID 3	322
Trace on Level	323
Chapter 20. String function blocks	325
Append Text	325
If	326
Is Empty	326
Is NULL	326
Left Fill	326
Left String	327

Lower Case	327
Object to String.	327
Repeat.	327
Replace	328
Right Fill	328
Right String	328
Substring by Position.	328
Substring by Value	329
Text Equal	329
Text Equal Ignore Case	329
Text Length	330
Trim Left	330
Trim Right	330
Trim Text.	330
Upper Case	330

Chapter 21. Utilities function blocks 333

Add Element	333
Catch Error	334
Catch Error Type	334
Condition	334
English	334
French.	334
German	335
Get Country.	335
Get Element	335
Get Language	335
Italian	336
Iterate Vector	336
Japanese	336
Korean	336
Loop	337
Move Attribute in Child.	337
New Locale	337
New Locale with Language	338
New Vector	338
Raise Error	338
Raise Error Type	338
Simplified Chinese	338
Size	339
To Array	339
Traditional Chinese	339

Part 4. Collaboration API reference 341

Chapter 22. BaseCollaboration class 343

dropFailedEvent()	343
dynamicSend()	344
existsConfigProperty()	345
getConfigProperty()	346
getConfigPropertyArray()	346
getCurrentLoopIndex()	347
getDBConnection()	347
getLocale()	349
getMessage()	350
getName()	351
implicitDBTransactionBracketing()	351
isCallerInRole().	352
isTraceEnabled()	352
logError(), logInfo(), logWarning()	353

queryFailedEvents()	355
raiseException()	357
resubmitFailedEvent()	360
saveFailedEvent()	362
sendEmail()	362
trace()	363
Chapter 23. BusObj class	367
copy()	368
duplicate()	369
equalKeys()	369
equals()	370
equalsShallow()	371
exists()	372
getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString()	372
getLocale()	374
getType()	375
getVerb()	375
isBlank()	376
isKey()	376
isNull()	377
isRequired()	378
keysToString()	378
set()	379
setDefaultAttrValues()	380
setKeys()	380
setLocale()	381
setVerb()	381
setWithCreate()	382
toString()	382
validData()	383
Deprecated method	384
Chapter 24. BusObjArray class	385
addElement()	386
duplicate()	386
elementAt()	387
equals()	387
getElements()	387
getLastIndex()	388
max()	388
maxBusObjArray()	389
maxBusObjs()	390
min()	391
minBusObjArray()	392
minBusObjs()	393
removeAllElements()	394
removeElement()	394
removeElementAt()	394
setElementAt()	395
size()	395
sum()	396
swap()	396
toString()	397
Chapter 25. CwDBConnection class	399
beginTransaction()	399
commit()	400
executePreparedSQL()	401

executeSQL()	402
executeStoredProcedure()	404
getUpdateCount()	405
hasMoreRows()	406
inTransaction()	406
isActive()	407
nextRow()	407
release()	408
rollBack()	409
Chapter 26. CwDBStoredProcedureParam class	411
CwDBStoredProcedureParam()	411
getParamType()	412
getValue()	413
Chapter 27. CxExecutionContext class	415
Static constants	415
CxExecutionContext()	415
getContext()	416
setContext()	416
Chapter 28. CxBiDiEngine class	419
BiDiBOTransformation()	419
BiDiBusObjTransformation()	420
BiDiStringTransformation()	421
Chapter 29. CollaborationException class	423
getMessage()	423
getMsgNumber()	424
getSubType()	424
getType()	425
toString()	426
Deprecated methods	427
Chapter 30. EventManagement package	429
EventKeyAttrDef()	429
EventQueryDef()	431
FailedEventInfo()	433
Chapter 31. Filter class	435
Filter()	436
filterExcludes()	437
filterIncludes()	438
recurseFilter()	439
recursePreReqs()	440
Chapter 32. Globals class	441
Globals()	442
callMap()	443
Chapter 33. SmartCollabService class	445
SmartCollabService()	445
doAgg()	446
doMergeHash()	446
doRecursiveAgg()	447
doRecursiveSplit()	447
getKeyValues()	448
merge()	448
split()	449

Chapter 34. StateManagement class	451
beginTransaction()	452
commit()	452
deleteBO()	452
deleteState()	453
persistBO()	453
recoverBO()	454
releaseDBConnection()	455
resetData()	455
retrieveState()	455
saveState()	456
setDBConnection()	456
StateManagement()	457
updateBO()	457
updateState()	457
<hr/>	
Part 5. Appendixes	459
Appendix. Standard Collaboration information	461
Standard processes for collaboration templates	461
Standard properties for collaboration templates	468
Notices	475
Programming interface information	476
Trademarks and service marks	477
Glossary	479
Index	483

Preface

IBM^(R) WebSphere^(R) InterChange Server and its associated toolset are used with IBM WebSphere Business Integration Adapters to provide business process integration and connectivity among leading e-business technologies and enterprise applications.

This document describes how to use Process Designer to create collaborations, which are part of the WebSphere InterChange Server infrastructure. Collaborations are programs that contain the business logic for application integration.

Audience

This document is for customers, consultants, or resellers who create or modify collaborations. Before you start, you should understand all the concepts explained in the manual *Technical Introduction to IBM WebSphere InterChange Server*.

To develop a collaboration, you should know standard programming concepts and practice. Also, collaboration development requires some Java[®] programming language knowledge. The collaboration API is based on the Java programming language, and it handles operations that most collaborations perform, such as manipulating business objects. If you have some programming background, the examples in this manual may help you to write simple collaborations, even if you do not know Java.

If your expertise is in applications and business analysis and you have never written a program, or if your collaboration needs to perform operations that are outside the scope of the collaboration API, you should work with a Java programmer.

Scope of this manual

The overall collaboration development process has many phases and can involve many people, including application experts, business analysts, and programmers. After analyzing an application integration problem, the collaboration development team builds the business process for solving it within the WebSphere business integration system. The team usually starts with a flow chart and migrates the flow chart to a collaboration.

This manual assumes that you are starting with a specification, flow chart, or pencil design. It does not cover analysis of business processes, development of connectors, or design of business objects.

Note: In this document backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes. All file pathnames are relative to the directory where the product is installed on your system.

How to use this manual

This manual is organized as follows:

Part I: Getting Started

Chapter 1, "Introduction to collaboration development," on page 3	Provides an overview of collaborations and the collaboration development environment.
Chapter 2, "Overview of Process Designer," on page 15	Provides detailed information about the Process Designer interface.

Part II: Creating a Collaboration Template

Chapter 3, "Designing a collaboration," on page 29	Provides information useful in the design phase of collaboration development.
Chapter 4, "Building a collaboration template," on page 81	Tells you how to create the definition of a collaboration template.
Chapter 5, "Using activity diagrams," on page 109	Describes how to use symbols and other components to build an activity diagram.
Chapter 6, "Using Activity Editor," on page 149	Describes how to use Activity Editor to graphically add activity definitions to a collaboration template.
Chapter 7, "Handling exceptions," on page 167	Describes how to implement exception handling in a collaboration template.
Chapter 8, "Workspace and layout options," on page 183	Describes some of your options for arranging the symbols in an activity diagram and the diagramming area itself.
Chapter 9, "Coding tips and examples," on page 193	Contains code snippets and tips that show how to perform common operations.
Chapter 10, "Creating a message file," on page 231	Explains how to set up the file that all collaborations need for holding logging and tracing messages.

Part III: Supported function blocks

Chapter 11, "Business object function blocks," on page 237	Describes the function blocks used to work with business objects.
Chapter 12, "Business object array function blocks," on page 261	Describes the function blocks used to work with business object arrays.
Chapter 14, "Database connection function blocks," on page 285	Describes the function blocks used to manage database connections.
Chapter 18, "Date function blocks," on page 311	Describes the function blocks used to work with dates.
Chapter 19, "Logging and tracing function blocks," on page 317	Describes the function blocks used to manage logging and tracing.
Chapter 20, "String function blocks," on page 325	Describes the function blocks used to work with String objects.
Chapter 21, "Utilities function blocks," on page 333	Describes the function blocks used to work with Vector objects and handle errors.

Part IV: Collaboration API reference

<p>Chapter 22, "BaseCollaboration class," on page 343; Chapter 23, "BusObj class," on page 367; Chapter 24, "BusObjArray class," on page 385; Chapter 25, "CwDBConnection class," on page 399; Chapter 26, "CwDBStoredProcedureParam class," on page 411; Chapter 27, "CxExecutionContext class," on page 415 Chapter 29, "CollaborationException class," on page 423; Chapter 31, "Filter class," on page 435; Chapter 32, "Globals class," on page 441; Chapter 33, "SmartCollabService class," on page 445; Chapter 34, "StateManagement class," on page 451</p>	<p>Contain reference pages for methods of classes in the collaboration API.</p>
<p>Glossary</p>	<p>Defines terms used in the manual.</p>

Related documents

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

You can install the documentation or read it directly online at one of the following sites:

- For InterChange Server documentation:
<http://www.ibm.com/websphere/integration/wicserver/infocenter>
- For collaboration documentation:
<http://www.ibm.com/websphere/integration/wbicollaborations/infocenter>
- For WebSphere Business Integration Adapters documentation:
<http://www.ibm.com/websphere/integration/wbiadapters/infocenter>

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

Note: Important information about this product might be available in Technical Support Technotes and Flashes issued after this document was published. These can be found on the WebSphere Business Integration Support Web site, <http://www.ibm.com/software/integration/websphere/support/>. Select the component area of interest and browse the Technotes and Flashes sections.

Typographic conventions

This document uses the following conventions:

<code>courier font</code>	Indicates a literal value, such as a command name, file name, information that you type, or information that the system prints on the screen.
bold	Indicates a new term the first time that it appears.
<i>italic</i>	Indicates a variable name or a cross-reference. When you view a document as a PDF file, cross references are both italic and blue. You can click on a cross-reference to jump to the target information.
<i>italic courier</i>	Indicates a variable name within literal text.
<code>boxed courier</code>	Separates a code fragment from the rest of the text.
<i>blue text</i>	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, brackets surround an optional parameter.
...	In a syntax line, ellipses indicate a repetition of the previous parameter. For example, <code>option[,...]</code> means that you can enter multiple, comma-separated options.
<i>ProductDir</i>	Represents the directory where the product is installed. For IBM WebSphere InterChange Server, the default product directory is <code>IBM\WebSphereICS</code> . In the IBM WebSphere Business Integration Adapters environment, the default product directory is <code>WebSphereAdapters</code> .

Summary of Changes

This chapter describes the following new features of IBM WebSphere InterChange Server, which are covered in this document.

New in release 4.3

September 2004

The descriptions of the function blocks have been enhanced. See Part 3, “Supported function blocks,” on page 235 for more information.

Support for bidirectional scripts has been added. See Chapter 3, “Designing a collaboration” and Chapter 28, “CxBiDiEngine class,” on page 419 for more information.

The Activity Editor has been enhanced. See Chapter 6, “Using Activity Editor,” on page 149 for more information.

The ability to automatically align scenarios in the Process Designer has been added. See “Aligning scenarios” on page 190 for more information.

The ability to configure for event sequencing or isolation has been added. See “Event sequencing” on page 56 and “Event isolation” on page 56 for more information.

Enhanced support for the recovery of failed events has been added. See “Handling failed events” on page 75, “dropFailedEvent()” on page 343, “queryFailedEvents()” on page 355, and “resubmitFailedEvent()” on page 360

Enhanced support for retrying a failed database connection has been added. See “Configuring database connection resilience feature” on page 72.

New in release 4.2.2

February 2004

The Activity Editor documentation has been greatly expanded. See Chapter 6, “Using Activity Editor,” on page 149 for information on using Activity Editor. In addition, all of the supported function blocks have been documented; see the chapters contained in Part 3, “Supported function blocks,” on page 235.

December 2003

The following changes have been made in this release:

- The Collaboration API documentation has been updated to include information on the Filter, Globals, SmartCollabServices, and StateManagement classes.
- You can now swap branches in a decision node. See “Swapping decision node branches” on page 125.
- Collaboration Debugger is a new tool that enables you to debug the business logic in a collaboration template. See “Debugging a collaboration” on page 108.

New in release 4.2

The following changes have been made in release 4.2:

- The CrossWorlds name is no longer used to describe an entire system or to modify the names of components and tools, which are otherwise mostly the same as in previous releases. For example, CrossWorlds System Manager is now System Manager, and CrossWorlds InterChange Server is now WebSphere InterChange Server.
- The CollaborationFoundation and WrapperFoundation templates have been included in this release. These templates help you create standard collaboration and wrapper collaboration templates for use in your business integration system. For more information on CollaborationFoundation and WrapperFoundation, see Chapter 3, “Designing a collaboration,” on page 29.
- All collaboration templates in release 4.2 are XML-based. The process for converting old templates (both XML and non-XML) to the format required by release 4.2 has been simplified. All templates stored in InterChange Server (ICS) are converted automatically when they are imported into a project in release 4.2. Templates stored in a file system (in other words, not in ICS) are converted automatically when they are imported into your Process Designer session. For more information, see “Updating templates from previous releases” on page 103.
- InterChange Server now supports long-lived business processes (LLBP); this feature enables collaborations to be deployed as a long-lived business process. Two asynchronous service calls have been added to support long-lived business processes: asynchronous inbound and asynchronous outbound. Both the new asynchronous inbound service call and the original synchronous service call can now specify a static or dynamic timeout value to further refine the parameters of a long-lived business process. See “Collaborations as long-lived business processes” on page 8 and “Designing for long-lived business processes” on page 54 for more information.
- Asynchronous service calls have been added to WebSphere InterChange Server. You can now implement an Asynchronous Outbound Service Call or an Asynchronous Inbound Service Call. To support the new asynchronous service calls, a new ServiceCall Properties Advanced Properties dialog box has been added; use this dialog box to configure the service calls. For more information, see “Service calls” on page 126.
- A decision node has been added to Process Designer’s diagram editor. This node replaces the conditional transition links and exception links used in previous versions. The decision node handles all branching tasks in a scenario diagram. For more information, see “Decision nodes” on page 120.
- The iterator node has been revised to accommodate For loops and While loops. In addition, a break symbol has been added to allow an iterator to terminate prematurely and return control to the parent diagram. See “Iterators” on page 139.
- The Process Designer GUI has been significantly enhanced:
 - The Service Call Properties dialog box has been redesigned to simplify the assignment of ports, verbs, and business object variables for both regular and compensation service calls. In addition, a new Advanced Service Call Properties dialog box has been added to enable you to specify the type of service call, its timeout value (if used with long-lived business processes), and its correlation attribute.
 - A Decision Properties dialog box has been added to support the new decision node. In addition, a new Decision Node symbol has been added to the Symbols toolbar.

- The Link Transition Properties dialog box has been modified to remove the conditional link functionality (replaced by the new decision node) and to add support for business object probes.
- The General tab of the Template Definitions window no longer contains a list of defined collaboration-specific properties. Instead, collaboration-specific properties can be added, modified, and viewed from the new Properties tab of the Template Definitions window.
- The Declarations tab of the Template Definitions window has been redesigned to simplify the process of declaring Java variables and importing Java packages. Variables and packages are declared and imported via a table; wherever possible, the table contains drop-down lists of all supported values to make it easier for the user to accurately declare and import needed items.
- You can now monitor business objects in an InterChange Server environment. By adding new business object probes to a transition link in a scenario, you can specify the particular objects you want to monitor, and view the results through the System Monitor.

Note: Business object probes cannot be used on the incoming transition link for a decision node or a service call link.

For more information, see “Using business object probes” on page 119.

- A new code fragment editor, called Activity Editor, has been added. This editor is used to add Java code fragments to action nodes; users can write traditional Java code or use it to drag and drop graphical elements to create an action node’s logic. The Activity Editor enables users to create action node logic without writing a full Java program. For more information, see “Adding activity definitions to an action node” on page 114.

Note: Activity Editor has not been fully documented for this release. Complete instructions for using Activity Editor will be available in a documentation patch release.

- Process Designer can export and import BPEL and UML files. You can import files for use as the basis for a new WebSphere InterChange Server collaboration template, or export a WebSphere InterChange Server collaboration template to BPEL or UML format for use with other applications. See “Converting templates” on page 102 for more information.
- Process Designer no longer requires a connection to ICS. All collaboration development operations can be done locally, within the context of a running System Manager session. The System Manager does not need to connect to a running ICS during collaboration development; the ICS connection is needed only when the collaboration is being deployed to the runtime environment.
- Information on creating an internationalized collaboration has been added to this book. See “An internationalized collaboration” on page 63.

New in release 4.1.1

This product has been internationalized.

New in release 4.1.0

The changes made in IBM CrossWorlds 4.0.1 do not affect the content of this document. However, the document has been modified to provide more description of the interaction between a triggering event and a collaboration has been expanded to include the interaction between a flow trigger and a collaboration. A

flow trigger is the object that triggers a collaboration. It consists of a business object and a verb (action). A flow trigger can be one of two types:

- A triggering event—when the flow trigger is sent by a connector agent
- A triggering access call—when the flow trigger is sent by an access client

Whether a flow trigger is a triggering event or a triggering access call is determined when the collaboration object is configured. The port of a collaboration template is configured for a particular business object type.

New in release 4.0.1

The changes made in IBM CrossWorlds 4.0.1 do not affect the content of this document. However, the document has been modified to restore the tutorial to the document. This tutorial creates a sample collaboration using Process Designer.

New in release 4.0.0

- The Process Designer interface has changed significantly. It now provides the ability to:
 - Create and modify collaboration templates locally (without being connected to the server).
 - Connect to the server.
 - Create and open collaboration templates directly (without using IBM CrossWorlds System Manager).
 - Create, edit, and save message text as part of the collaboration template. Message content is now saved in the repository. During template compilation, Process Designer generates a message text file that is used only at runtime.
 - Save all collaboration template information (including message content) as a single text file on the client machine.
 - Compile all or a specified subset of collaboration templates.
 - Set display preferences, such as modifying activity diagram colors, enabling and disabling automatic display of a start node in a blank new diagram, and enabling and disabling the display of connection points on nodes in an activity diagram.
 - Convert collaboration templates to the new format; you can specify conversion either on an individual template, a specified subset of templates, or all templates.

Menus have been added and changed to accommodate the above changes.

- The collaboration runtime environment now provides the ability for a collaboration:
 - To execute queries in a database. For more information, see “Executing database queries” on page 212 and “getDBConnection()” on page 347.
 - To identify a service call exception caused by transport failure. For more information, see “Service calls and exactly-once requests” on page 178 and `ServiceCallTransportException` in “getSubType()” on page 424.
 - When the destination connector has been configured for parallel processing and a service call fails, to determine that the request was queued up in the agent master, but never got dispatched to the application; therefore, the request may be resent. For more information, see `AppRequestNotYetSent` in “getSubType()” on page 424.

- The collaboration runtime environment prevents deletion of a collaboration template if an existing collaboration object has been built from it. For more information, see “Deleting a collaboration template” on page 107.
- The Test Connector has been enhanced. To prevent the Java process on your machine from running out of memory when you run the Test Connector with a large business object, causing the connector to crash, the connector’s batch file includes a command that increases the memory to 128 Mb. You can further increase the memory allocation.

In addition, this document has been modified in the following ways:

- Includes the following subtypes of `ServiceCallException`:

- `AppBusObjDoesNotExist`
- `AppRetrieveByContentFailed`
- `AppRequestNotYetSent`

For more information, see “`getSubType()`” on page 424.

- Includes a more detailed explanation of the different syntax forms for `raiseException()`.
- Because Process Designer has been significantly enhanced, the tutorial chapter that appeared in the previous version of this manual is obsolete and has been removed. A new tutorial may appear in a subsequent update of this manual. Check the IBM CrossWorlds Documentation Update Report for revisions to this guide. Alternatively, if you are reading this document online, right-click on its title on the title page for direct access to the most up-to-date version.

Note: The tutorial for the new version of Process Designer has been added to the document. For more information, see “New in release 4.0.1” on page xx.

- The chapter, “Distributing a Collaboration Template”, has been removed from this manual. For information on moving collaboration templates between servers, see the *System Implementation Guide*.
- The process for converting a collaboration template to the new format is described in “Converting templates” on page 102.

Part 1. Getting started

Chapter 1. Introduction to collaboration development

Welcome to Process Designer and to the collaboration development process. Process Designer is a powerful modeling and code generation tool with which you can create collaborations, programs that create enterprise business processes that involve multiple applications.

This chapter is an introduction to the collaboration development process and the tools used to develop collaborations.

What are collaborations?

Collaborations are software modules that describe business processes and that run within IBM® WebSphere® InterChange Server (ICS). These business processes are programs that contain the business logic for application integration. A collaboration can perform various types of Java™ operations. However, most commonly, collaborations perform operations on business objects, such as:

- Obtaining and manipulating one or more values in the triggering event
- Sending a business object as a request to an application so that the application creates, deletes, or updates a specified entity
- Sending a request to an application to retrieve an entity

As Table 1 shows, a collaboration is a two-part entity, consisting of a repository definition and a runtime object.

Table 1. Parts of a collaboration

Repository entity	Runtime object
Collaboration template	Collaboration object

When you install a collaboration, you install a *collaboration template*. A collaboration template contains all of the collaboration's execution logic, but it is not executable. To execute a collaboration, you must first create a *collaboration object* from the template. The collaboration object becomes executable after you configure it by binding it to connectors or to other collaboration objects, and specifying other configuration properties.

Note: For an introduction to how collaborations function as a component of the IBM WebSphere InterChange Server system, see the chapter on collaborations in the *Technical Introduction to IBM WebSphere InterChange Server*. This section concentrates on defining a collaboration in terms of how it is developed.

In this book, both collaboration templates and collaboration objects are often referred to as simply collaborations, unless it is necessary to distinguish between a template and an object.

Collaboration templates

A collaboration begins as a collaboration template. A collaboration template is a specification of the logic within the collaboration. You define a collaboration

template with the Process Designer tool, which stores the appropriate information in System Manager. Development of a collaboration template involves the following steps:

- Creating the collaboration template
- Building the parts of the collaboration template
- Compiling the collaboration template

Creating a collaboration template

When you develop a collaboration, you use a tool called Process Designer to develop a collaboration template. Process Designer provides an easy-to-use, graphical user interface (GUI) that eliminates much of the coding usually required to develop a program. This interface makes it easy for you to declare variables, write code fragments, and so on. IBM WebSphere InterChange Server also provides generic collaboration templates to facilitate the development process. Using Process Designer, it is simpler to develop a collaboration template than to write a standard programming language program. However, the end result of collaboration development is a program, in the form of a Java class.

Process Designer saves the collaboration template information in System Manager until deployment. After a collaboration is deployed, the collaboration information is available in InterChange Server, where it can be accessed when the collaboration receives a triggering event. For more information on Process Designer, see Chapter 2, “Overview of Process Designer,” on page 15.

Building a collaboration template

Within Process Designer, building a collaboration template involves a two-level development process:

- Activity diagrams
 - You create activity diagrams, which are graphical, symbolic descriptions of the business process and its flow.
 - You use Activity Editor to implement additional details of the business process.

Compiling the template converts the diagrams and their associated code to an executable Java class.

- Messages—You define messages, which hold the text used in logging, tracing, and raising exceptions.

When a template is compiled, the message content is placed in a message file within System Manager. After the collaboration is deployed, the message file is also stored in InterChange Server, where it is accessed during runtime. For more information, see Chapter 10, “Creating a message file,” on page 231.

Creating the activity diagrams: A collaboration template consists of:

- *scenarios*, which specify sets of actions.

When developing a collaboration template, you first divide the collaboration’s business logic into one or more scenarios.

Every collaboration template is partitioned into one or more units called scenarios. A scenario specifies exactly how the collaboration responds to a particular flow trigger. The scenario is like a method in that it describes the actions that will be taken by the collaboration.

You can create multiple scenarios or put all of the collaboration’s logic into one scenario.

- *activity diagrams*, which describe these actions using code fragments representing individual actions.

For each scenario, you create an activity diagram that graphically describes the scenario's process. An activity diagram is a graphical implementation of the scenario, including actions, execution flow, and external calls. Activity diagrams are based on Unified Modeling Language (UML), a standard notation for modeling business processes. The use of visual programming in diagrams makes it easy to create a scenario and reduces the amount of actual coding.

The various steps of the activity diagram are the individual actions, or code fragments.

Every scenario contains at least one, top-level diagram that represents the entry point of the scenario during execution and contains the overall logic flow of the scenario. *Subdiagrams* can divide the details of scenario logic into multiple nested levels.

An activity diagram looks somewhat like a flow chart. Unlike a flow chart, however, an activity diagram can create the executable Java code that the activity diagram represents.

Figure 1 is an example of an activity diagram.

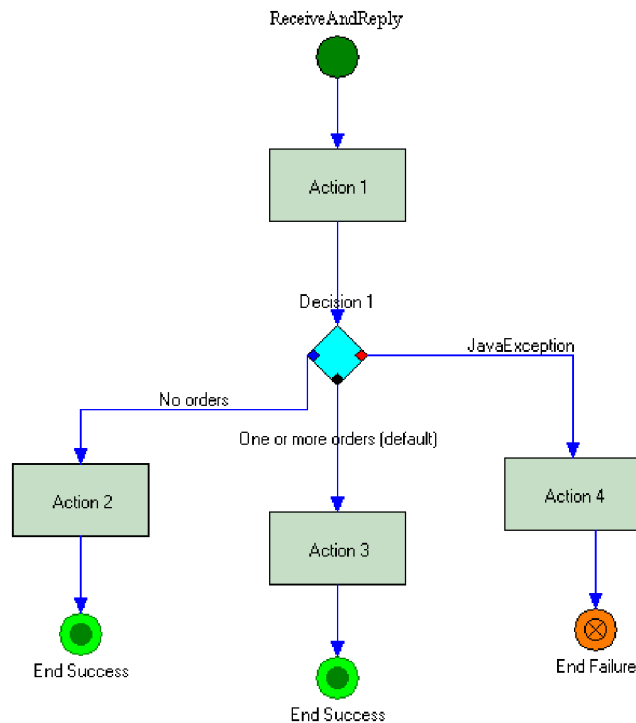


Figure 1. Activity diagram

The basic unit in an activity diagram is an *action*, represented by a rectangle. An action specifies a unit of work in the collaboration and is used to create and store Java code fragments.

The activity diagram represents all the possible behavior at execution time. The activity diagram in Figure 1 has multiple execution paths. An *execution path* is represented by a set of symbols and links that flow from the top Start symbol to one of the End symbols at the bottom. A symbol that has multiple outgoing links is a decision node; it is at this point that the collaboration decides to follow one path of logic instead of another.

Implementing Java code fragments: Each action contains a Java programming language *code fragment* to which a developer can add more code. Process Designer embeds the code fragment in the collaboration template code that it generates and executes the generated code when the collaboration object executes, as part of the collaboration flow.

You can add a Java code fragment, if desired. You can:

- Write your own code.

Much of the code in a collaboration template consists of calls to the WebSphere InterChange Server *collaboration API*. However, the code fragment is not limited to calls to the collaboration API. It can contain any Java code that you want to write. Add your own activity definition or customize an existing activity definition by doing one of the following:

- Using Activity Editor, a GUI that facilitates adding activity definitions. It enables you to either write traditional Java code or graphically model the programming logic with function blocks.
- Directly typing the Java code fragment into the Code Fragment window of the action node’s Action Properties dialog box. (Note that you must set Process Designer user preferences to enable this feature.)

- Import code from another Java class.

You can import external packages of Java classes into the collaboration and use their methods inside actions.

Note: The class that Process Designer generates must run in the execution context of ICS. Although you can import or write your own Java code, the code should augment an activity diagram. Performing other operations that can destroy the flow of execution or consume excessive resources is discouraged.

Compiling a collaboration template

When you finish the definition of a collaboration template—you have defined its scenarios, built the activity diagrams, customized its code fragments, and created its message file—you compile the entire template. The collaboration compilation process creates three types of files (.class, .java, and .txt) that the collaboration runtime uses.

When you compile a collaboration, these files are automatically created in your Integration Component Libraries (ICL) project within System Manager. When you deploy your collaboration object to the server, these files are moved into the *productDir*\collaborations directory. Table 2 describes the files and shows where they are located after compilation and deployment.

Table 2. Collaboration files

File type	Description	Location
.class	Final executable class file that Process Designer produces during compilation	After compilation: <i>ICLProject</i> \Templates\Classes After deployment: classes\UserCollaborations
.java	Source code file that Process Designer produces during code generation	After compilation: <i>ICLProject</i> \Templates\Src After deployment: classes\UserCollaborations

Table 2. Collaboration files (continued)

File type	Description	Location
.txt	Message file that contains all of the message text you added to the template during development	After compilation: <i>ICLProject\Templates\messages</i> After deployment: <i>\messages</i>

Important

Make all changes to messages only through Process Designer; never make changes directly to the message text file. After a collaboration has been deployed, this file is used by the runtime environment; directly editing it can cause errors.

After you have compiled a collaboration template, you can use System Manager to create collaboration objects and deploy these objects and the template to InterChange Server. See *Implementation Guide for WebSphere InterChange Server*.

Collaboration objects

Although a collaboration template contains the collaboration’s execution logic, you must take the following steps before the collaboration can execute:

1. Create a collaboration object.

A collaboration object is an instance of a collaboration template. To create a collaboration object, you use System Manager.

2. Configure the collaboration object.

The collaboration object becomes executable after you configure it. To configure the collaboration object, you bind it to connectors or to other collaboration objects, and specify other configuration properties

The process of specifying the objects with which a collaboration object interacts is called *binding*. A collaboration object can be bound to any of the following:

- A connector, other collaboration objects, or access clients with which a collaboration object interacts. When you bind the collaboration object and specify the values for its configuration properties, the collaboration object becomes executable.

For more information on using System Manager to create and configure collaboration objects, see the *Implementation Guide for WebSphere InterChange Server*.

Figure 2 illustrates the creation of a collaboration object called OrderStat from the template OrderStatus.

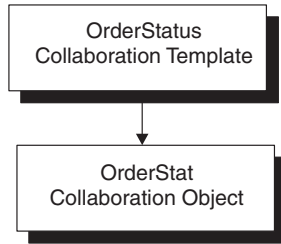


Figure 2. Creating a collaboration object

The OrderStatus collaboration template was created with two defined ports, through which the collaboration expects to communicate with its source and destination objects. As part of configuring the OrderStat collaboration object, you bind it to two external objects. Figure 3 shows that the OrderStat collaboration object is bound to the SAP connector and to the Vantive connector.

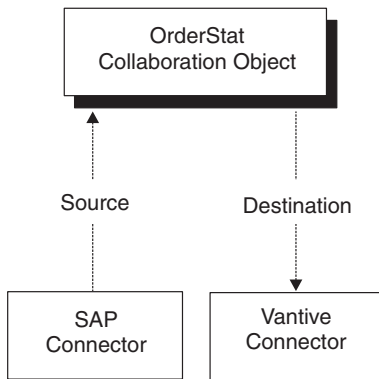


Figure 3. Collaboration object bound to connectors

After a collaboration object is bound and configured, you can use System Manager to test it and deploy it in the runtime environment. A collaboration object can be configured to run in one thread or in multiple threads, with each thread handling one triggering event. For concurrent handling of multiple triggering events, you run a collaboration object in multi-threaded mode.

Collaborations as long-lived business processes

Collaboration objects can be deployed as long-lived business processes, enabling asynchronous communication between business processes. As a result, business processes can span a greater length of time. In a long-lived business process, the event flow context (including global template or port variables and business object variables created in Process Designer, as well as runtime workflow information) persists throughout a service call.

Service call timeout values can be specified for asynchronous inbound and synchronous service calls to further define the parameters of a long-lived business process.

If you plan to use a collaboration object as a long-lived business process, you must configure the collaboration template accordingly. Before building your collaboration template, see the information in “Designing for long-lived business processes” on page 54. After you have designed your template, see Chapter 4, “Building a collaboration template,” on page 81 for information about the specific configuration tasks required to provide support for long-lived business processes.

Collaborations and the IBM WebSphere InterChange Server system

The InterChange Server business integration system uses a business object to carry data and action requests from one application to another. A collaboration begins execution when a scenario within a collaboration object receives a particular business object and an action (verb). This combination of business object and verb whose receipt by the collaboration triggers the execution of a scenario is called a *flow trigger*. As part of the design of the collaboration template, the collaboration developer specifies the business objects (and verbs) that act as flow triggers for each scenario. As part of the configuration of the collaboration object, you bind the incoming port of the collaboration to a particular source for the flow trigger. The type of source that sends the flow trigger to the incoming port determines the type of flow trigger that the collaboration receives.

As Table 3 shows, a flow trigger can be one of several types, based on the source of the incoming business object:

Table 3. Types of flow triggers

Flow trigger	Source of incoming business object
Triggering event	Connector or another collaboration
Triggering access call	Access client (through the Server Access Interface within ICS)

Note: An access client is an external process that can request execution of collaborations through the Server Access Interface API. For more information, see the *Access Development Guide*.

Because connectors are the most common source of flow triggers, the term “triggering events” is often used to refer to the incoming business objects of a collaboration. For example, the Template Definition window includes a tab called Ports and Triggering Events. From this tab, you can define collaboration ports and assign triggering events to its scenarios. However, even though the titles of this tab and the associated table within this tab include the term “triggering events”, this tab handles assignments of either type of flow trigger: triggering events or triggering access calls. If the scenario receives its business object from a connector, its flow trigger is a triggering event (as the name of the tab indicates). If the scenario receives its business object from an access client, its flow trigger is a triggering access call. In this case, you would still use the Ports and Triggering Events table to assign a triggering access call to a scenario.

The type of flow trigger for the collaboration is not actually determined until the port of the collaboration object is configured:

- Internal port—when the port is bound to a connector, it receives its business object in the form of a triggering event.
- External port—when the port is bound to an access client, it receives its business object in the form of a triggering access call.

For more information on how to configure a collaboration object, see the *Implementation Guide for WebSphere InterChange Server*. For more information on the Ports and Triggering Events tab of the Template Definition window, see “Defining ports and triggering events (the Ports and Triggering Events tab)” on page 94.

Tools for collaboration development

The platform for collaboration development is Windows® 2000. Collaborations are written in Java. Table 4 lists the tools that WebSphere InterChange Server provides for collaboration development.

Table 4. Tools for collaboration development

Tool	Description	For more information
Process Designer	Graphical tool that assists in the development of the collaboration template.	"Process Designer"
IBM WebSphere InterChange Server Collaboration API	Set of Java classes with which you can customize the generated collaboration code. (The methods in the API can also be accessed through the Activity Editor function blocks.)	"Collaboration API" on page 10
System Manager	Tool that provides graphical windows to create and configure a collaboration object.	"System Manager" on page 11
Integrated Test Environment (Test Connector)	A suite of tools used to test business processes. Use the Test Connector tool (available in the Integrated Test Environment and as a standalone tool) to simulate a generic connector so you can easily test a collaboration's design.	"Test Connector" on page 12

Process Designer

Process Designer is used for creating, editing, compiling, and deleting collaboration templates. When modifying an existing template, you can use Process Designer to edit the template's properties, as well as to add or edit scenarios and activity diagrams.

For detailed information about the Process Designer interface, see Chapter 2, "Overview of Process Designer," on page 15.

Collaboration API

The WebSphere InterChange Server collaboration API provides several classes whose methods you can use in a collaboration template.

Note: You can access the API both through traditional Java calls and through the supported function blocks in the Activity Editor. See Chapter 6, "Using Activity Editor," on page 149 for more information.

Interacting with a collaboration object

The BaseCollaboration class generically defines the behavior and functions of a collaboration, such as obtaining the values of configuration properties, writing messages to a log file, and tracing.

When you create a collaboration template, you create a Java class that is a subclass of BaseCollaboration. As such, your collaboration inherits all of the methods of BaseCollaboration. These methods allow a collaboration to perform operations such as:

- Get the value of a configuration property
- Raise an exception
- Write informational, warning, and error messages to a log file

For more information on the methods of the BaseCollaboration class, see Chapter 22, "BaseCollaboration class," on page 343.

Interacting with business objects

A collaboration generally interacts with and manipulates business objects. Methods of the BusObj class enable a collaboration to perform operations such as:

- Get the name of a business object
- Get the key values of a business object
- Get the number of child business objects contained in a hierarchical business object
- Test whether the attribute values of two business objects are equal
- Copy attribute values from one business object to another

For more information on the methods of the BusObj class, see Chapter 23, “BusObj class,” on page 367.

Interacting with business object arrays

Collaborations frequently get and set the values of business object attributes. When a business object is hierarchical, one or more of its attributes is a child business object, or perhaps an array of child business objects. A child business object appears as an array to the collaboration.

Methods on the BusObjArray class let a collaboration interact with and manipulate business object arrays. These methods perform operations such as:

- Set or get elements of the array
- Copy an array to another array
- Add a business object to the array
- Get the number of elements in the array

For more information on the methods of the BusObjArray class, see Chapter 24, “BusObjArray class,” on page 385.

Interacting with exceptions

When errors occur in a collaboration, the collaboration or the collaboration runtime environment raises an exception. The exception is contained in an object of the CollaborationException class. This class lets a collaboration object interact with an exception object and perform the following operations:

- Get the exception type or subtype
- Get the exception message

For more information on the methods of the CollaborationException class, see Chapter 29, “CollaborationException class,” on page 423.

System Manager

System Manager is a graphical tool that provides an interface to ICS and its repository. It enables you to do the following collaboration-related tasks:

- Create a collaboration object
- Bind a collaboration object
- Set collaboration-specific properties of a collaboration object
- Test a collaboration’s design (through the Test Connector tool)
- Deploy a collaboration object to the runtime environment

For more information on how to use System Manager to create, configure, and deploy a collaboration object, see the *Implementation Guide for WebSphere InterChange Server*.

Test Connector

The Test Connector is a graphical tool for testing collaborations and connectors. It is available both in the Integrated Test Environment and as a standalone tool.

Note: If you are testing access clients, you must use Test Connector through the Integrated Test Environment.

The Test Connector tool simulates an actual connector, allowing you to easily test the design of your collaborations by sending in a triggering event or sending a service call request. For more information on how to use Test Connector, see the *Implementation Guide for WebSphere InterChange Server*.

Overview of the development process

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

1. Install and set up the IBM WebSphere InterChange Server software (including the Java Development Kit and all other required third-party products). See the WebSphere InterChange Server system installation guide for your platform for specific installation and configuration instructions.
2. Design and implement the collaboration.

Stages of collaboration development

The stages of collaboration development are as follows:

1. Design the business process that the collaboration will implement.
2. Create the business object definitions.
3. Create the collaboration template, including meta-information and definitions.
4. Create each scenario and its activity diagram.
5. Customize any required code fragments.
6. Create the message text.
7. Compile the template.
8. Create a collaboration object from the collaboration template.
9. Test and debug the collaboration.
10. Deploy the collaboration to the runtime environment.

Table 5 on page 13 provides a visual overview of the collaboration development process and a quick reference to chapters where you can find information on specific topics.

Note: Some of the overall collaboration development tasks fall outside the somewhat more narrow scope of developing a collaboration template, and therefore are not documented in this guide. For each of these tasks, Table 5 on page 13 provides a reference to the appropriate document in the WebSphere InterChange Server library.

Note that if a team of people is available for collaboration development, the major tasks of developing a collaboration can be done in parallel by different members of the development team.

Table 5. Overview of the collaboration development tasks

Task:	Steps:	Refer to:
Design collaboration architecture	<ul style="list-style-type: none"> • Identify business processes • Investigate use of collaboration templates • Customize for special business flows 	Chapter 3, "Designing a collaboration"
Design and develop business objects	<ul style="list-style-type: none"> • Design structure of business objects • Implement business objects 	Business Object Development Guide
Build the collaboration template	<ul style="list-style-type: none"> • Create the collaboration template and edit its properties 	Chapter 4, "Building a collaboration template"
	<ul style="list-style-type: none"> • Create the scenarios and activity diagrams 	Chapter 5, "Using activity diagrams"
	<ul style="list-style-type: none"> • Customize the code fragments 	Chapter 6, "Using Activity Editor"
	<ul style="list-style-type: none"> • Implement error and message handling 	Chapter 7, "Handling exceptions"
Create the collaboration object	<ul style="list-style-type: none"> • Create collaboration definition • Customize runtime features, if any 	Implementation Guide for WebSphere InterChange Server
Test and debug	<ul style="list-style-type: none"> • Test collaboration in IBM WebSphere InterChange Server System • Recode as needed 	Implementation Guide for WebSphere InterChange Server

Chapter 2. Overview of Process Designer

Process Designer enables you to perform the following collaboration development tasks:

- Create, edit, compile, or delete a template definition through the Template Definitions window.
- Define or edit an activity diagram for a scenario of the collaboration template through the diagram editor.

This chapter provides an introduction to Process Designer. It describes the interface and how to navigate through the Process Designer windows, menus, and toolbars to perform the tasks required for collaboration development.

Starting Process Designer

The method by which you start Process Designer can vary depending on whether you are creating a new collaboration template or editing an existing collaboration template.

Important

Before you can start Process Designer, you must ensure that System Manager is running.

There are several ways to start Process Designer from within System Manager, as described in Table 6.

Table 6. Starting Process Designer from within System Manager

Method	Result
Right-click the Collaboration Templates folder from the object browser view, then click Create New Collaboration Template from the context menu.	Process Designer opens and displays the New Template dialog box.
Double-click a collaboration template within the Collaboration Templates folder	Process Designer opens and displays the template definition you double-clicked.
Create and use a user project shortcut to the component in the Integration Component Libraries	Process Designer opens and displays the template definition associated with the shortcut.

You can also launch Process Designer from the Start menu. Click Start —> Programs —> IBM WebSphere InterChange Server —> IBM WebSphere Business Integration Toolset —> Development —> Process Designer.

Process Designer displays in its own dockable window. You can launch more than one Process Designer instance at a time to edit more than one collaboration template definition.

Process Designer layout

When you start Process Designer, by default the main window is displayed as shown in Figure 4.

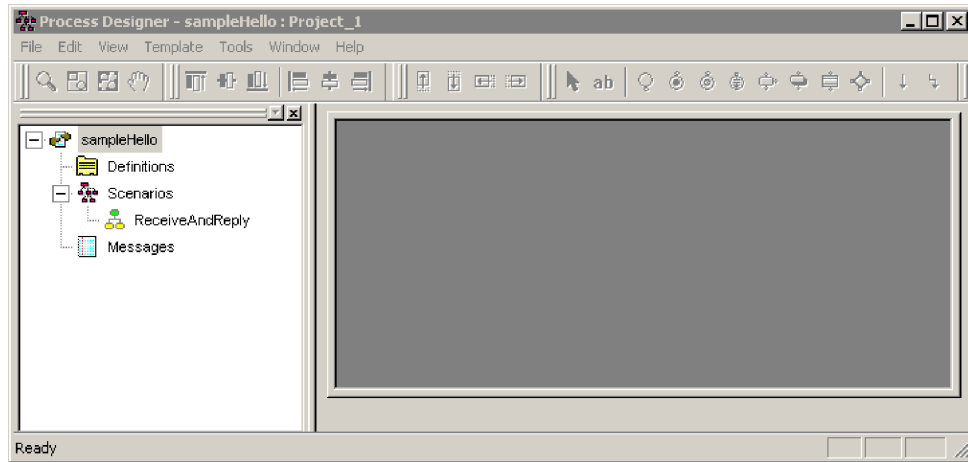


Figure 4. Process Designer main window

The layout of the Process Designer window consists of the following areas:

- Template tree (dockable)

The template tree view in the left pane uses a hierarchical format to list the collaboration template's definitions, scenarios, and messages. Click the plus sign (+) next to an existing scenario node in the tree to expand its subtree and view its existing scenarios and subdiagrams, if any.
- Working Area of the Main Window, which can be blank or it can display the following:
 - Template Definitions window

This window is used to provide general information about the collaboration template, variable declarations, or port information. See “Template Definitions window” on page 17 for more information.
 - Diagram editor window

This window is used to display the nodes of the activity diagram. For more information, see “Diagram editor window” on page 18.
 - Template Messages window

This window is used to write or edit the template's message file. For more information, see “Template Messages window” on page 19.

The Template Definitions and Template Messages windows and the Diagram Editor can be minimized, maximized, and sized (opened to a user-specified size) within the working area. For more information, see “Displaying windows within the working area” on page 25.
- Compile output window (dockable)

The compile output window (often called just the output window) displays results from the compilation of a collaboration template. Process Designer automatically displays this window when you compile the collaboration template. For more information, see “Compiling a collaboration template” on page 101.

Note: Process Designer stores the configuration of its main window when it exits. Therefore, any changes you make to this configuration display when you next open Process Designer. (For more information, see “Customizing the main window” on page 23.) Figure 4 shows the default configuration of the main window. If previous invocations of Process Designer have changed this configuration, your main window can be different.

You can access Process Designer’s functionality in any of the following ways:

- Pull-down menus at the top of the window
- Icons in the toolbars
- Context-sensitive menu (a popup menu accessed through a right-mouse click)
- Keyboard shortcuts

Process Designer windows

Template Definitions window

The Template Definitions window provides four tabs for defining collaboration properties.

- General tab—provides fields in which you specify general information about the collaboration template, such as its name, description, minimum transaction level, and package.
- Declarations tab—provides fields in which you specify variable declarations.
- Properties tab—provides fields in which you specify the name, type, and value for user-defined collaboration template properties.
- Ports and Triggering Events tab—provides fields in which you specify port names and their associated business objects and verbs.

Important

IBM recommends that you do not add, modify, or delete a business object to or from the repository using Business Object Designer or System Manager after you have bound the business object to a collaboration object running in a production environment. For more information see “Defining ports and triggering events (the Ports and Triggering Events tab)” on page 94.

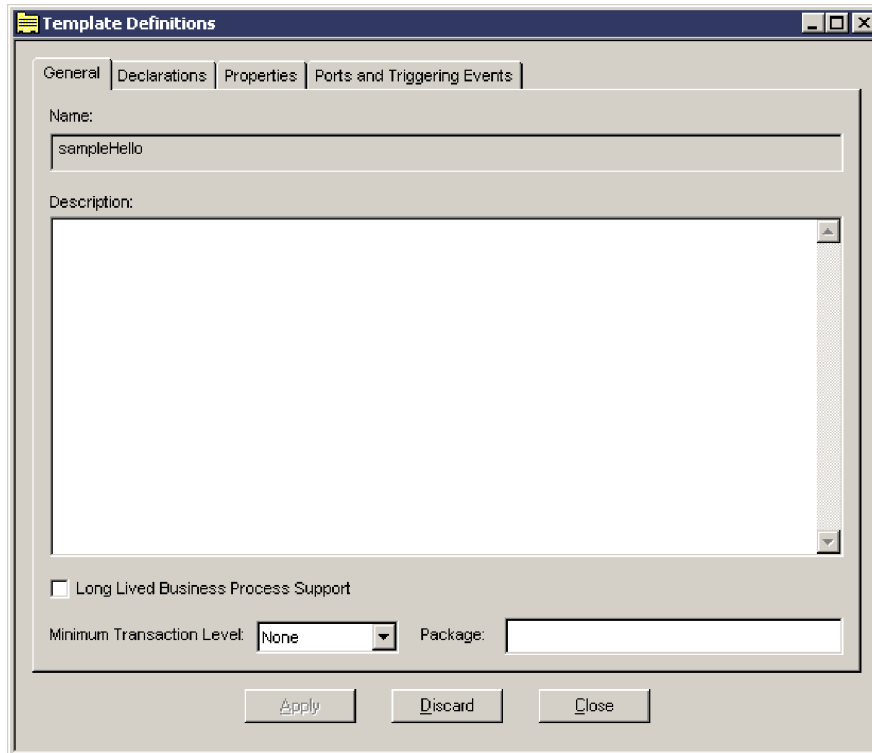


Figure 5. Template Definitions window

There are several ways to open the Template Definitions window, which displays in the working area of the main window:

- In the template tree view, double-click Definitions.
- In the template tree view, select Definitions, right-click and choose Open Template Definitions.
- From the Template pull-down menu, choose Open Template Definitions.
- Use the shortcut key combination Ctrl+T.

The Template Definitions window contains Apply and Discard buttons; these buttons appear at the bottom of the window regardless of which tab is currently displayed. The Apply button commits the changes to the template, but does not save them (you must use the File → Save command to save all changes). The Discard button lets you revert to the previously saved definition, discarding changes that you have not yet saved.

Note: The Discard and Apply buttons affect the data contained in all tabs, not just the tab that is currently visible.

Diagram editor window

The diagram editor is a tool within Process Designer that enables you to create and edit activity diagrams. This window displays in the working area of the main window when you open an activity diagram. In the diagram editor, you can add nodes, service calls, and transition links to an activity diagram; change the placement of items; add and edit text labels and fonts; and add and change individual component properties.

There are several ways to open the diagram editor:

- In the template tree view, expand the Scenarios node and double-click on the name of a diagram; alternately, right-click on the name and choose Open Diagram.
- In the template tree view, expand the Scenarios node, select the name of a diagram, and choose Open Diagram from the Template pull-down menu.
- From the Template pull-down menu, choose Open All Diagrams.

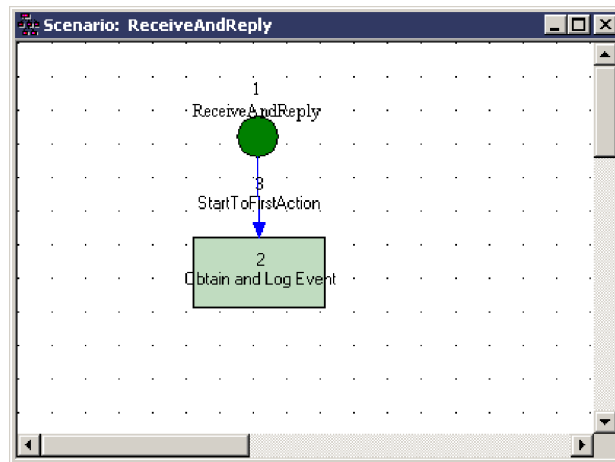


Figure 6. Diagram editor window

For more information on how to use the diagram editor, see “Creating an activity diagram” on page 100.

Template Messages window

The Template Messages window provides an area in which you can write or edit the template’s message file. When you compile the template, the message text is written to the appropriate Integration Component Library project’s Template\messages directory within System Manager.

There are several ways to open the Template Messages window:

- In the template tree view, double-click Messages.
- In the template tree view, select Messages, right-click and choose Open Messages.
- From the Template pull-down menu, choose Open Template Messages.
- Use the shortcut key combination Ctrl+M.

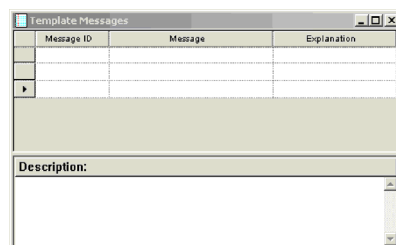


Figure 7. Template Messages window

Process Designer menus

In Process Designer, the enabled menus and menu options depend on what displays in the Working Area. The following sections explain the main menus of Process Designer when the Working Area is empty or displays the Template Definitions window:

- “Functions of the File menu”
- “Functions of the View menu” on page 21
- “Functions of the Template menu” on page 22
- “Functions of the Window menu” on page 23

Note: When the Working Area displays the diagram editor, Process Designer enables options in the Edit menu, and enables different options in some of the other menus. Most of these functions pertain to working with activity diagrams, and are discussed in “Accessing diagram editor functionality: Process Designer menus” on page 109.

Functions of the File menu

When the Working Area is empty or it displays the Template Definitions or Template Messages window, the File menu displays the following options:

- New—Creates a new collaboration template.
- Open—Opens an existing collaboration template definition. Contains the following two options:
 - From Project—Opens a collaboration template from an Integration Component Library user project.
 - From File—Opens a collaboration template from a .cwt file stored in the file system.
- Close—Closes the collaboration template.
- Save—Saves the current collaboration template. Contains the following two options:
 - To Project—Saves a collaboration template to an Integration Component Library user project.
 - To File—Saves a collaboration template to a .cwt file that is stored in the file system.
- Save As—Saves the current collaboration template under a different name. Contains the following two options:
 - To Project—Saves a collaboration template to an Integration Component Library user project.
 - To File—Saves a collaboration template to a .cwt file that is stored in the file system.
- Delete—Displays the Delete template from Project’ dialog box, from which you can choose the collaboration template to delete.
- Compile—Compiles the collaboration template. For more information, see “Compiling a collaboration template” on page 101.
- Compile All—Enables you to compile all collaboration templates in your project, or to specify a subset for compilation. For more information, see “Compiling multiple collaboration templates” on page 101.
- Import—Imports files into the template definition. The Process Designer Importer can import BPEL and UML (in XMI format) files; the files are converted as necessary to WebSphere InterChange Server 4.2 template files.

- **Export**—Exports files. You can export a template file to UML (in XMI format) or BPEL format. The Process Designer Exporter tool performs all necessary format conversions.
- **Exit**—Closes Process Designer.

When the Working Area displays the diagram editor, the File menu displays the additional Page Setup, Print Preview, and Print options to enable printing of an activity diagram. When the Working Area displays the Template Messages window, the File menu displays an additional option that allows you to print the message file.

Functions of the Edit menu

The Edit menu options are available only when the diagram editor is active. Options include standard Windows edit commands (for example, Undo, Redo, Copy, and Paste) and the following special Process Designer options:

- **Select All**—Selects all nodes in the current activity diagram.
- **Find ID**—Finds the activity diagram ID.
- **Find Text**—Lets you find text in the current activity diagram.
- **Replace Text**—Lets you find and replace text in the current activity diagram.
- **Properties**—Lets you edit the properties of a selected symbol. This option is enabled only when a symbol is selected in the workspace.
- **Font**—Lets you change the font and color of text labels of selected symbols in an activity diagram. You can change the font of the currently selected symbols and links or you can change the font of all components by first using the Select All option to select all components in a diagram, then applying the font change. This option is only activated when a symbol is selected in the workspace.

Functions of the View menu

The View menu functions are valid when Process Designer first opens and when the Working Area display pertains to the visual appearance of activity diagrams. Many of these functions can be toggled on or off:

- **Preferences**—Opens the User Preferences dialog box, which enables you to specify how items are represented in Process Designer.
- **Template Tree**—When this option is on, Process Designer displays the template tree view as the left pane of the Process Designer window.
- **Output Window**—When this option is on, Process Designer displays the results of the template compilation.
- **Toolbars**—Controls display of the different toolbars of Process Designer. The submenu options include:
 - **Standard**—When this option is on, Process Designer displays the buttons for the Standard toolbar.
 - **Symbols**—When this option is on, Process Designer displays the buttons for the Symbols toolbar.
 - **Align**—When this option is on, Process Designer displays the buttons for the Alignment toolbar.
 - **Nudge**—When this option is on, Process Designer displays the buttons for the Nudge toolbar.
 - **Zoom**—When this option is on, Process Designer displays the buttons for the Zoom/Pan toolbar.
 - **Programs**—When this option is on, Process Designer displays the buttons for accessing other WebSphere InterChange Server programs.

- **Status bar**—When this option is on, Process Designer can display its single-line status message at the bottom of the main window.

In addition, Process Designer enables the following diagram options when the diagram editor is active:

- **View Types**—When on, displays a symbol’s type. This option is useful to help you learn to recognize a node by its shape.
- **View UIDs**—When on, displays the unique ID (UID) of each symbol.
- **View Labels**—When on, displays the user-provided symbol label.
- **Lock (Read only)**—When on, puts the activity diagram into read-only mode.
- **Refresh**—Refreshes the activity diagram display.
- **Grid**—When on, displays the workspace grid lines. When off, grid lines are hidden.
- **Snap to Grid**—When on, new symbols are automatically aligned with the grid lines when they are placed in the activity diagram.
- **Grid Properties**—Lets you set the grid properties. (Note that Angle Snap is not applicable to activity diagrams, though it can be toggled on and off.)
- **Page Bounds**—Shows the page boundaries as dashed lines.
- **Zoom commands**—Lets you enlarge the activity diagram or zoom to one section. You can also perform zoom commands from the Zoom toolbar. For more information on zooming, see “Zooming or panning on symbols” on page 186.
- **Auto Layout**—Automatically aligns the scenario open in the diagram editor.
- **Template Auto Layout**—Automatically aligns all scenarios inside a template.

Functions of the Template menu

When the Working Area is empty or it displays the Template Definitions or Template Messages windows, the Template menu displays the following options:

- Whenever any object except a scenario is selected in the template tree view:
 - **Open All Diagrams**—Opens all activity diagrams defined for the collaboration template.
 - **Close All Diagrams**—Closes all open activity diagrams.
 - **New Scenario**—Displays the New Scenario dialog box.
 - **Open Template Definitions**—Displays the Template Definitions window, from which you can modify properties of the collaboration template.
 - **Open Template Messages**—Displays the Template Messages window, from which you can define or modify the message file associated with the collaboration template.
- Whenever a scenario is selected in the template tree view, the following additional menu items are available:
 - **Open Diagram**—Opens the activity diagram for the current scenario.
 - **Rename Scenario**—Enables you to rename the current scenario.
 - **Delete Scenario**—Deletes the current selected scenario and its activity diagrams.
 - **Open Scenario Definition**—Enables you to edit scenario-level variables.
- When the diagram editor is open, the following menu items are available in addition to those already described:
 - **Size Diagram**—Resizes the activity diagram in units of vertical and horizontal page counts, and is relevant for printing the activity diagram. The Diagram

size dialog contains spin controls for numeric page inputs. Note that diagram size is directly related to paper orientation (landscape or portrait) and paper selection.

- Save Diagram as Text File—Saves the current activity diagram to a file in a text format (.txt).

Functions of the Tools menu

The Tools menu enables you to launch other WebSphere InterChange Server tools. The options are as follows:

Map Designer	Opens Map Designer.
Business Object Designer	Opens Business Object Designer.
Relationship Designer	Opens Relationship Designer.

Functions of the Window menu






The Window menu pull-down options encompass the standard Multiple Document Interface (MDI) window display functions. Use these options to control display features such as tiling, cascading, and activating open windows.

Process Designer toolbars

Process Designer provides toolbars with common tasks you need to perform. These toolbars are dockable; that is, you can detach them from the palette of the main window and float them over the main window or the desktop.

Table 7 lists the toolbars that Process Designer provides.

Table 7. Process Designer toolbars

Toolbar name	Toolbar appearance	For more information
Standard		None
Symbols		“Introduction to the symbols” on page 110 “Aligning symbols” on page 183
Alignment		“Aligning symbols” on page 183
Nudge		“Nudging symbols” on page 185
Zoom/Pan		“Zooming or panning on symbols” on page 186

Customizing the main window

Process Designer provides the following ways to customize its main window:

- Choose which windows display
- Float a dockable window
- Choose how windows within the working area display

Choosing windows to display

As Figure 4 shows, when you first open Process Designer, the template tree view displays in the left pane. The working area displays on the right and is empty. The output window does not display. You can customize the appearance of the main window options from the View menu.

Table 8 describes the options of the View pull-down menu and how they affect the appearance of the Process Designer main window.

Table 8. View Menu options for main window customization

View Menu option	Element displayed
Template Tree	The template's definitions, scenarios, and messages as the left-hand pane.
Output Window	The output window as a small window under the template tree view (if it displays) and the working area.
Toolbars	A menu that provides options for displaying the Process Designer toolbars: <ul style="list-style-type: none"> Standard The main toolbar in the Process Designer palette, which provides buttons that allow you to connect to or disconnect from ICS, open a template from the Server or from a file, save and compile a template, cut, copy, paste, and delete a template, and print. Symbols The Diagram Symbols toolbar provides the symbols to add to an activity diagram. Align The Alignment toolbar contains alignment features for activity diagram symbols. Nudge The Nudge toolbar contains features that slightly move selected symbols of an activity diagram. Zoom The Zoom/Pan toolbar contains features that zoom or pan selected symbols of an activity diagram.
Status Window	A single-line pane in which Process Designer displays status information

When a menu option appears with a check mark to the left, the associated element displays. To turn off display of the element, choose the associated menu option. The check mark disappears to indicate that the element does not currently display. Conversely, you can turn on display of an undisplayed element by choosing the associated menu option. In this case, the check mark appears beside the displaying element.

Floating a dockable window

Process Designer supports the following portions of the main window as dockable windows:

- Template tree view
- Output window
- Toolbars

By default, a dockable window is usually placed along the edge of the main window and moves as part of the main window. When you float a dockable window, you detach it from the main window, allowing it to function as an independent window. To float a dockable window, hold down the left mouse button, grab the border of the window and drag it onto the main window or desktop.

Displaying windows within the working area

The working area of the Process Designer main window allows you to display windows within it in any of the following ways:

- **Maximized**—One window takes up the entire working area. You can switch between maximized windows by choosing the name of the desired window from the Windows pull-down menu. If you have kept the default user preferences for Workbook Diagram Windows, you can also switch between maximized windows by choosing the corresponding Workbook tab below the working area. For more information, see “Changing general display” on page 187.
- **Sizable**—Each window is its own separate area within the working area. You can resize these windows and can cause them to overlap and move within the working area. Sizable windows are useful when you want simultaneous display of more than one activity diagram or an activity diagram and the Template Definitions or Template Messages window.
- **Minimized**—Each window is represented as an icon at the bottom of the working area. You can restore a minimized window by double-clicking its minimized representation.

Part 2. Creating a collaboration template

Chapter 3. Designing a collaboration

This chapter describes design guidelines meant to help you achieve reusable, well-behaved collaborations.

In general, it is good practice to develop a standard collaboration template to facilitate development of user-defined collaborations. Use of such a template ensures:

- Consistency of collaboration design
When based on a standard template, your collaborations can all:
 - Perform the same verb operations on the business object in the destination application that corresponds to the collaboration’s flow trigger
 - Handle errors using the same error handling mechanism, greatly simplifying the technical support of the final collaborations
 - Use ports with identical names, types, and expected behavior
- Simplicity of documenting the collaboration
Documenting the behavior of the collaborations based on the standard template can become much simpler because it too can be based on a template of information that the user needs to understand the collaboration’s behavior.
- Incorporation of “best practices”
Best practices that IBM recommends (see “Coding recommendations” on page 35) and that your own site develops can be incorporated into the standard template and automatically included into collaborations based on this standard template.

WebSphere InterChange Server provides a standard collaboration template (CollaborationFoundation) and a standard wrapper collaboration template (WrapperFoundation) that you can use as-is, or modify to meet your requirements. See “CollaborationFoundation template” and “WrapperFoundation template” on page 47 for more information.

In addition, this chapter provides guidelines for the following tasks:

- Creating collaboration object groups. See “Building collaboration groups” on page 52 for more information.
- Handling parallel execution, including event sequencing and event isolation. See “Designing for parallel execution” on page 55 for more information.
- Creating an internationalized collaboration template. See “An internationalized collaboration” on page 63 for more information.
- Handling failed events. See “Handling failed events” on page 75 for more information.

CollaborationFoundation template

The CollaborationFoundation collaboration template is a tool that facilitates the development of user-defined collaborations that perform functions such as the following:

- Synchronization

A collaboration performs synchronization when it replicates data from one application to another. Such a synchronization typically processes data from a source application that does not expect a result.

CollaborationFoundation contains the verb logic, data filtering, and error handling used by most collaborations that perform synchronization; this is the basic synchronization logic. CollaborationFoundation has five empty subdiagrams, each named Additional Processing, that enable you to extend the collaboration's business process without modifying the base synchronization logic.

This section primarily describes the basic synchronization logic and offers recommendations for extending the CollaborationFoundation template to meet the requirements of the collaborations you plan to develop.

- Data access

A collaboration can be triggered by a Retrieve request from a source application that waits for the result. A collaboration can also be triggered by a source application that sends a synchronization event and waits for the collaboration to return a full business object from the destination application. Such events require a source application that can send a synchronous request or that uses the Server Access Interface.

This chapter describes the access functionality in Retrieve Process and Additional Retrieve Process.

- Data piping

A collaboration performs data piping when it moves data from one application to another without performing any filtering or verification of the data. To build such a collaboration from CollaborationFoundation, bind the From port to the source application, and then bind both the DestinationAppRetrieve and To ports to the destination application.

Note: You can also modify a product-delivered collaboration to simply move data from a source to a destination. Bind the From port to the source application, bind both the DestinationAppRetrieve and To ports to the destination application, and bind all other ports to the port connector. It is important that you do not change the default value of any configuration property.

The following sections provide additional information about the CollaborationFoundation template:

- "CollaborationFoundation features"
- "Using the CollaborationFoundation template" on page 33
- "CollaborationFoundation ports" on page 34
- "Extending CollaborationFoundation" on page 35

CollaborationFoundation features

A collaboration object generated from the CollaborationFoundation template can perform a Create, Retrieve, Delete, or Update action on a business object in the destination application that corresponds to the collaboration's triggering business object.

Handling the triggering business object

Behavior of the Retrieve verb: CollaborationFoundation is designed to enable the triggering business object to use the Retrieve verb. If the collaboration is bound to a source connector that makes a synchronous request, the triggering business

object's values are returned to the source connector as soon as the collaboration completes its processing. The values are returned as if the business object's values had been passed by reference.

This template provides the standard properties shown in Table 9; these properties affect the behavior of business flows.

Table 9. CollaborationFoundation standard properties for business flow

Standard property	Description
CONVERT_CREATE	Configures a collaboration object to convert the verb sent to the destination from Create to Update if the triggering business object already exists in the destination application.
CONVERT_UPDATE	Configures a collaboration object to convert the Update verb to the Create verb if the business object does not already exist in the destination application.
USE_RETRIEVE	Configures a collaboration object to retrieve its triggering business object from the destination application before it synchronizes data. This property is useful when performing compensation processing and when setting the verb based on whether the business object already exists in the destination application.
ADDITIONAL_RETRIEVE	Configures a collaboration object to retrieve the business object from the destination application after it synchronizes data. This property is useful when the source application requires a full-valued business object to be returned from the destination application, but the destination application's connector does not return a complete business object after creating or updating its data.

Figure 8 on page 32 shows the main business flow for a collaboration when the USE_RETRIEVE property evaluates to false.

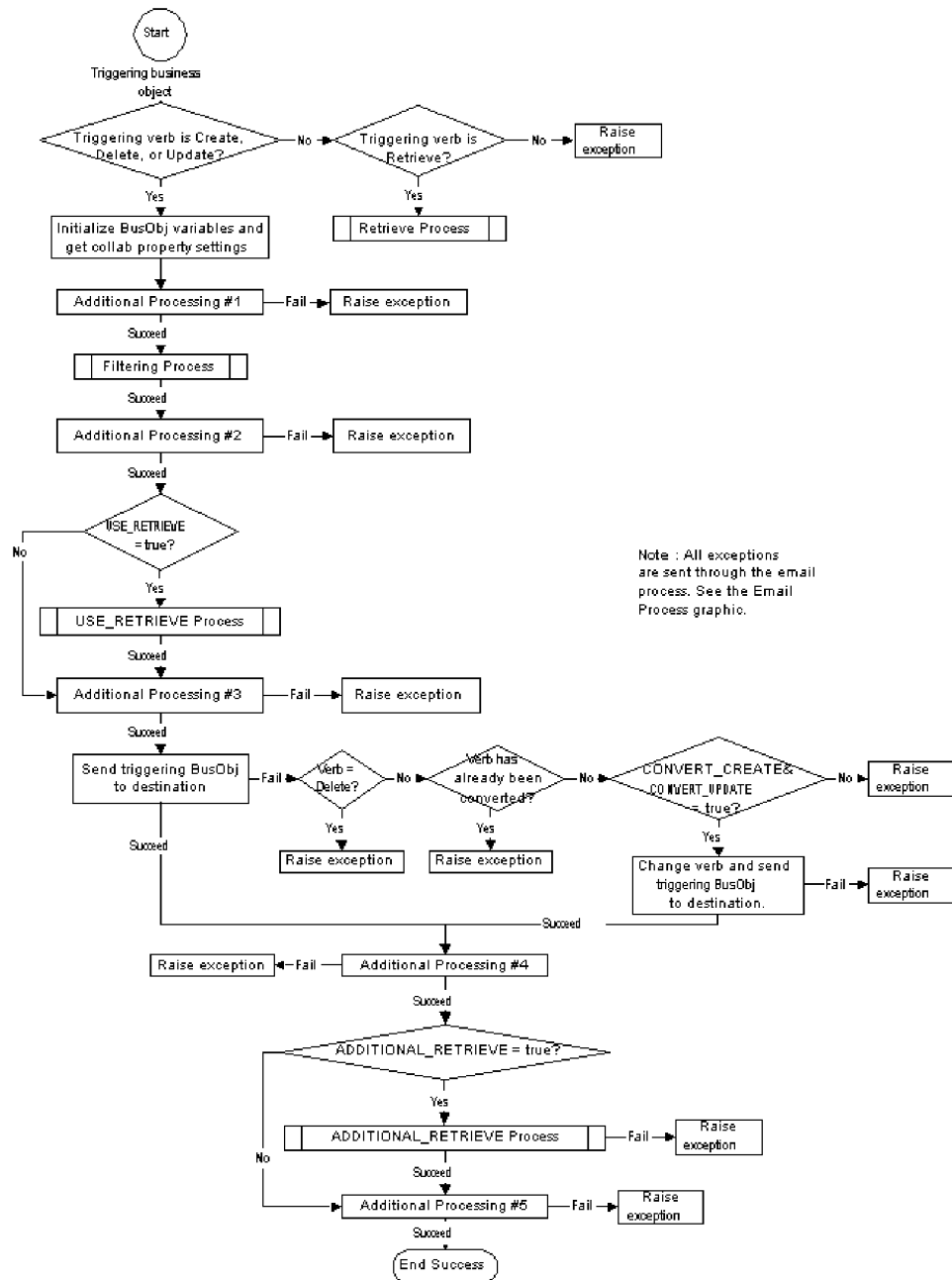


Figure 8. CollaborationFoundation main business-process flow

Filtering data in the triggering business object: You can configure a collaboration object to filter data in specified attributes of the triggering business object. You can use the results of the filtering to determine whether the collaboration needs to synchronize a triggering business object with specific data. The template provides the following four properties for filtering data:

- 1_FILTER_ATTRIBUTE
- 1_INCLUDE_VALUES
- 1_EXCLUDE_VALUES
- 1_FAIL_ON_INVALID_VALUE

For information on adding additional properties that can specify multiple filtering attributes, see the description of the 1_FILTER_ATTRIBUTE property in “Standard

Collaboration information,” on page 461. In addition, Figure 93 on page 466 illustrates the collaboration’s filtering behavior.

To filter on a second attribute, the CollaborationFoundation template provides the following set of properties:

- 2_FILTER_ATTRIBUTE
- 2_INCLUDE_VALUES
- 2_EXCLUDE_VALUES
- 2_FAIL_ON_INVALID_VALUE

To filter on multiple attributes, provide a set of filter properties for each property. Provide a unique number for each set of properties. In other words, the filter properties for the third attribute can use 3_ as the prefix. Although the prefix must be different for each set of attributes, the property names themselves must be identical to those shown above.

Error handling

CollaborationFoundation provides standard properties for error handling. Table 10 describes these properties.

Table 10. Standard properties for error handling

Error-handling property	Description
INFORMATIONAL_EXCEPTION	Specifies whether a collaboration object must end successfully and send the exception to a trace, or end in failure and raise an exception to be logged.
SEND_EMAIL	Specifies whether a collaboration object sends email to a specified address when an exception occurs, regardless of the value of INFORMATIONAL_EXCEPTION.

“Email process for error handling” on page 467 illustrates the collaboration’s email process for error handling.

For both of the error-handling properties, you can specify the behavior for all exceptions or for a smaller set of comma-delimited message numbers. The message numbers correspond to those in the collaboration’s message file (collaborations\messages\CollaborationFoundation.txt).

Using the CollaborationFoundation template

Perform the following tasks to use the CollaborationFoundation template:

1. Copy the entire CollaborationFoundation template to your development area.
2. Rename the collaboration template to reflect the collaboration you are building.
3. Within Process Designer, use the Ports and Triggering Events tab of the Template Definitions window to change the type of the existing ports (DestinationAppRetrieve, To, From) to assign the appropriate business objects to them. For more information, see “CollaborationFoundation ports” on page 34.
4. Create additional sets of filter properties if you want your collaboration to filter more than one business object attribute.

WebSphere InterChange Server provides a set of configuration properties that enable you to include or exclude business objects from synchronization based on specified values in a specified attribute. If you want to filter synchronization based on values in more than one attribute, you must create an additional set

of the configuration properties for each attribute you want to evaluate. For more information, see “Standard properties for collaboration templates” on page 468.

5. If necessary, create additional collaboration functionality by adding code to the Additional Processing subdiagrams. These sub diagrams have no functionality in the CollaborationFoundation template. They exist to enable easy customization of the collaboration.
6. Save the collaboration template and compile it.

CollaborationFoundation ports

Figure 9 illustrates CollaborationFoundation’s ports. The tables that follow provide information about each port.

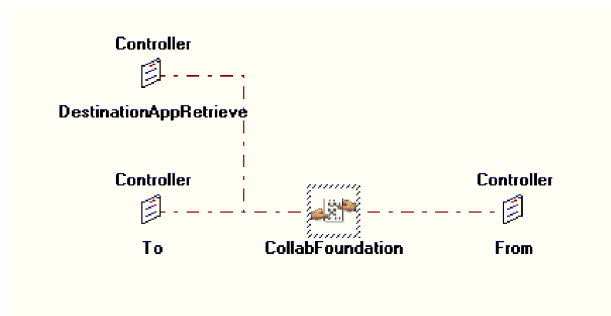


Figure 9. CollaborationFoundation collaboration ports

The DestinationAppRetrieve port

Table 11 lists the features of the CollaborationFoundation’s DestinationAppRetrieve port.

Table 11. Port features (DestinationAppRetrieve port)

Port feature	Value
Business Object	A reference-valued BusObj. The default value is Controller.
Verbs Used	Retrieve
Compensating Verbs	None
Purpose	Retrieves the full-valued business object from the destination and uses it to set the verb.
Bound To	Destination application’s connector
Configuration Property	USE_RETRIEVE

The To port

Table 12 lists the features of the CollaborationFoundation’s To port.

Table 12. Port features (To port)

Port feature	Value
Business Object	A full-valued BusObj. The default value is Controller.
Verbs Used	Create, Update, or Delete
Compensating Verbs	None
Purpose	Sends a business object out of the collaboration. Often the collaboration sends a copy of the triggering business object through this port to the destination.

Table 12. Port features (To port) (continued)

Port feature	Value
Bound To	Destination application's connector or another collaboration (for additional processing)
Configuration Property	None

The From port

Table 13 lists the features of the CollaborationFoundation's From port.

Table 13. Port features (From port)

Port feature	Value
Business Object	triggeringBusObj (the default value is Controller)
Verbs Used	Create, Retrieve, Update, or Delete
Compensating Verbs	None
Purpose	Receives the triggering business object
Bound To	Source application or a calling collaboration
Configuration Property	None

Extending CollaborationFoundation

This section contains the following sections to help you customize collaborations based on CollaborationFoundation:

- "Coding recommendations"
- "Common modifications" on page 39

Coding recommendations

This section describes coding practices to help you standardize your code with that in product-delivered collaborations.

- "Naming conventions"
- "Processing the flow trigger"

Naming conventions

It is good practice to establish naming conventions for use in your collaboration templates. The following list provides some naming conventions:

- Identify each variable's type by prefixing its name with a meaningful letter. For example, prefix a String variable's name with the letter "s"; prefix a Boolean variable's name with the letter "b". The following code initializes two such variables:

```
String sExceptionType
Boolean bBranch
```

- Collaboration configuration properties should be in all uppercase letters to easily distinguish them from program variables. The following code obtains the value of the SEND_EMAIL property:

```
bSendEmail = getConfigProperty("SEND_EMAIL");
```

Processing the flow trigger

Process Designer automatically declares a variable of type BusObj called triggeringBusObj. This variable holds the flow trigger (usually a triggering event), which caused the scenario to execute.

Several situations might require you to work with the flow trigger even after it has been through processing such as having data added to it after being sent out through service calls, or having the values of attributes manipulated. Such situations include the following:

- Sending the flow trigger out through service calls to perform a rollback during the compensation steps of a transactional collaboration
- Comparing the values of attributes in the flow trigger with the values of attributes in a business object returned by a service call or database lookup

To handle these situations, it is recommended that you create an intermediate BusObj variable that is a copy of the flow trigger, then manipulate the intermediate variable and send it out through service calls as necessary rather than modify the flow trigger.

Note: Creating copies of business objects consumes system resources. If your business process does not require an intermediate variable (because there are not transactional requirements, and you do not ever have to compare the values of attributes before and after certain situations, for instance), use the flow trigger rather than a copy of it to preserve resources.

If your collaboration is configured to be a long-lived business process, however, the content of the flow trigger business object (triggeringBusObj) is not preserved across service calls. In this case, always make a copy of the triggering flow.

There are several APIs available that enable you to copy the contents of one business object into another and each has advantages and disadvantages; the sections “Using the copy() method” and “Using the duplicate() method” address each approach.

Using the copy() method: The copy() method can be used to copy the contents of one business object variable into another business object variable of the same type. It is recommended that you take this approach because the collaboration templates delivered by WebSphere InterChange Server do so, and having consistency between delivered and custom-built components results in greater maintainability.

To follow this approach you must instantiate a new BusObj object of the same type as the triggering business object; it is recommended that you perform the instantiation in the scenario definition of the scenario and that you name the variable that stores the copy processingBusObj. To satisfy these requirements and recommendations, add the following line of code to the scenario definition of the scenario:

```
BusObj processingBusObj;  
processingBusObj = triggeringBusObj.duplicate();
```

Next you must run the copy() method on the processingBusObj variable and pass the triggeringBusObj variable to it as an argument. It is good practice to do this in the first action node of the top-level diagram of the scenario—one that you dedicate exclusively to initializing variables. The example code below copies the contents of the triggeringBusObj variable into the processingBusObj variable:

```
processingBusObj.copy(triggeringBusObj);
```

Using the duplicate() method: For example, the code fragment below declares a variable of the same type as the flow trigger and sets its values by duplicating the values in the business object of the flow trigger:

```
BusObj processingBusObj;  
processingBusObj = triggeringBusObj.duplicate();
```

The collaboration uses `processingBusObj` to manipulate data as required. When it is ready to send the data to the destination application, the collaboration copies the intermediate variable to the `ToBusObj` variable. It uses `ToBusObj` in its service call to the destination application. The code fragment below shows the statement that copies the data to `ToBusObj`:

```
ToBusObj.copy(processingBusObj);
```

After the service call returns successfully to the collaboration, the collaboration copies `ToBusObj`'s values to `triggeringBusObj`, as shown below:

```
triggeringBusObj.copy(ToBusObj);
```

WebSphere InterChange Server collaborations do not generally change the original value of `triggeringBusObj` until the collaboration has received the returned `ToBusObj` from the To port. Using the intermediate variable ensures that the collaboration changes the value of `triggeringBusObj` only after successfully receiving values from the destination application.

Raising exceptions

Catch exceptions at the level at which they occur, then raise them to the top process in the collaboration. By catching the exception, you can specify how to handle the exception and control how it appears to the user; for example, you can make clear the context in which the exception occurred. Moreover, creating action nodes for exception handling provides visual documentation of each place in the code where exceptions can occur.

In any collaboration, you must raise each trapped exception until it reaches the collaboration runtime environment. If you use a service call that triggers another collaboration, the calling collaboration must check for exceptions as a result of the service call.

To raise exception text to a calling diagram, declare separate string variables to store the message text and the exception type. For example, the following code declares two such string variables:

```
String sMessage  
String sExceptionType
```

Use branching to provide different behavior when the service call succeeds or fails. In the branch that handles failure, assign values into the two string variables. For example:

```
sMessage = currentException.getMessage();  
sExceptionType = currentException.getType();
```

Before returning control to the process that made the service call, raise the exception. For example:

```
raiseException(ServiceCallException, 4000, SendRefBusObj.getType(),  
    SendRefBusObj.getVerb(), SendRefBusObj.keysToString(),  
    sExceptionType, sMessage);
```

The code above specifies error message 4000, which is the standard error message for collaboration failure. The message file includes the following text:

```
4000
Collaboration Failed: {1}.{2} with keys ({3}) synchronization failed
and the exception is {4}.{5}.
[EXPL]
The business object could not be synchronized in the destination.
```

In the preceding text, the `raiseException()` method substitutes the values shown in Table 14.

Table 14. Substituted values in `raiseException()` call

Variable	Substituted text
{1}	The value that <code>SendRefBusObj.getType()</code> returns
{2}	The value that <code>SendRefBusObj.getVerb()</code> returns
{3}	The value that <code>SendRefBusObj.keysToString()</code> returns
{4}	The value in the <code>sExceptionType</code> variable
{5}	The value in the <code>sMessage</code> variable

If the process that makes the service call is *not* the topmost process in the collaboration, the process making the service call must raise the exception to its calling process. Each process above the calling process must also raise the exception so that the error message can be logged from the topmost process.

Branching

The flow of a collaboration diagram is often based on the value of a collaboration configuration property. The collaboration can use the property value to set a boolean variable, which it later uses to determine which path to take. For example, the following code declares and initializes a boolean variable named `bBranch`:

```
boolean bBranch = false;
```

WebSphere InterChange Server sets the value of the branching variable according to conditions in the code. These conditions may be based on the value of several boolean variables. For example, suppose the collaboration evaluates its `CONDITION_TWO` property only if its `CONDITION_ONE` property evaluates to `true`.

The code below bases a branch on the value of two boolean variables:

- `bCondition1`, which contains the value configured for the collaboration's `CONDITION_ONE` property
- `bCondition2`, which contains the value configured for the collaboration's `CONDITION_TWO` property

This code sets the value of `bBranch` to `true` if `CONDITION_ONE` evaluates to `true` and `CONDITION_TWO` evaluates to `false`; it sets the value of `bBranch` to `false` if `CONDITION_ONE` evaluates to `false` or `CONDITION_TWO` evaluates to `true`:

```
if (bCondition1 && !bCondition2)
{
    bBranch = true;
}
else
{
    bBranch = false;
}
```

Wrapper collaborations

A *wrapper collaboration* is a collaboration that handles the verification or synchronization of a business object for another collaboration. The calling collaboration sends a top-level business object that is referenced on its own flow trigger to the wrapper collaboration.

For example, a `SalesOrderProcessing` collaboration can synchronize the generic Order business object. Generic Order contains references to a generic Customer business object, which represents the customer making the order. Moreover, generic Order contains an array of generic `OrderLineItem` business objects. Each `OrderLineItem` references a generic Item business object, which represents the items ordered.

To modularize collaboration logic, you can provide separate collaboration templates to process generic Order and the generic business objects that it references. For example, to process an Order that references Customer and Item business objects, you can provide the following templates:

- `SalesOrderProcessing`—processes the order.
- `CustomerWrapper` and `CustomerSync`—process the referenced customer.
- `ItemWrapper` and `ItemSync`—process the referenced items.

Separating business object processing into different, specific collaborations not only enhances the reusability of each collaboration template, but also prevents two collaborations from modifying the same data at the same time. For more information, see “Problems in concurrent processing” on page 55.

Common modifications

This section discusses the following common modifications to the `CollaborationFoundation` code:

- “Handling dependent data”
- “Delegating the checking of dependent data” on page 40
- “Performing recursive checking of dependent data” on page 44

Use the modifications presented in this section to standardize your customized collaborations with those delivered with the product.

Handling dependent data

A business object is said to have dependent data when it references another business object that affects its own processing. For example, assume that you want to synchronize Order, Customer and Item business objects at your site. Before your `SalesOrderProcessing` collaboration object synchronizes each Order, it needs to ensure that the Customer referenced by the Order already exists in the destination application. Furthermore, you want it to ensure that each Item referenced by each `OrderLineItem` also exists in the destination application, and perhaps even halt the Order’s synchronization until you have verified that its referenced business object exists. In a case like this, the Order object is dependent upon the existence of its referenced Customer and Item objects.

Because WebSphere InterChange Server delivers several collaborations whose triggering business objects may reference a Customer or an Item business object, a single installation sometimes simultaneously runs more than one collaboration object that is trying to synchronize the same Customer or Item data. To maintain data consistency and prevent multiple collaborations from concurrently modifying the same instance of the same type of business object, delegate synchronization of

referenced business objects to separate collaboration objects. Collaborations call intermediate wrapper collaborations to handle the verification or synchronization of business objects referenced on their triggering business objects. Wrapper collaborations facilitate data isolation in environments that run multiple collaborations.

Use wrapper collaborations to maintain data consistency in an environment that runs multiple collaborations. For example, assume that two CustomerSync collaboration objects run on the same installation. One runs in conjunction with SalesOrderProcessing, while the other runs independently. In this scenario, the independent CustomerSync collaboration object synchronizes Customer data between the same two applications as the SalesOrderProcessing collaboration object. If you use SalesOrderProcessing to delegate the synchronization of reference-valued Customer business objects to the intermediate CustomerWrapper collaboration object, the software provides event isolation. In other words, the system does not allow the standalone CustomerSync collaboration object to work on the same business object at the same time as the CustomerSync collaboration object called by SalesOrderProcessing.

For more information on synchronizing business objects referenced by a triggering business object, see “Problems in concurrent processing” on page 55.

Delegating the checking of dependent data

You can extend CollaborationFoundation to delegate a dependency check to another collaboration. To do so, you must perform the following tasks:

- Add a port that is going to be bound to the called collaboration.
- Add a collaboration configuration property that enables you to specify whether the collaboration verifies or synchronizes referenced business objects. If you opt to synchronize referenced Item or Contact business objects, you might need to create additional configuration properties.
- Add messages that provide information if the verification or synchronization fails.
- Add a scenario to process the dependency check.

Adding a port: To delegate dependent data, create a port for each dependent business object. Maintain consistency with other product-delivered collaborations by naming the new port *ToBONameWrapper*, where *BOName* refers to the dependent, reference-valued business object. For example, SalesOrderProcessing uses a port called *ToCustomerWrapper* to send reference-valued Customer business objects to CustomerWrapper.

Create one such port for each reference-valued business object to be sent to a wrapper collaboration. For example, SalesOrderProcessing also uses ports named *SendContactRef* and *SendItemRef*.

Adding a configuration property: To delegate checking of a referenced business object to another collaboration, create a new collaboration configuration property. This property enables the person configuring the collaboration to specify whether the collaboration is going to verify or synchronize the referenced business object.

To maintain consistency with the product-delivered collaborations, name the new property *VERIFY_SYNC_BOName*, where *BOName* is the reference-valued business object. For example, SalesOrderProcessing contains a property called *VERIFY_SYNC_CUSTOMER*, which it uses to delegate checking of a Customer business object. Establish the possible values of this property as follows:

- `neither`—Prevents delegation.
- `sync`—Causes the collaboration to send the reference-valued business object with the `Sync` verb to the appropriate wrapper collaboration for synchronization.
- `verify`—Causes the collaboration to send the reference-valued business object with the `Exists` verb to the appropriate wrapper collaboration for verification.

By default, the system sets a `VERIFY_SYNC_BOName` property to `neither`.

Error-handling properties: When a collaboration delegates an array of reference-valued business objects, it must be prepared to handle the synchronization or verification failure of any element in the array. The product-delivered collaborations provide configuration properties that enable you to specify how to handle the failure of individual elements of an array during verification or synchronization.

For example, the generic `Order` business object contains an array of `OrderLineItem` child business objects; each of these `OrderLineItem` objects can reference an `Item` business object. Therefore, the collaboration that handles the `Order` business object can delegate an array of `Item` business objects. Similarly, the generic `Order` business object contains an array of `OrderContactRef` child business objects, and each `OrderContactRef` can reference a `Contact` business object. Therefore, the collaboration handling the `Order` business object can delegate an array of `Contact` business objects.

The `SalesOrderProcessing` collaboration, which handles `Order` business objects, provides configuration properties that enable you to specify what happens when one or more referenced business objects in an array fail synchronization or verification.

One option is to stop synchronization of the triggering business object if any business object in a delegated array of referenced business objects fails synchronization or verification. Another option is to remove the failed business object from the array and continue synchronization of the triggering business object.

A collaboration evaluates options for handling a failed reference-valued business object only if it is attempting to verify or synchronize that business object. Therefore, the collaboration needs to evaluate a property that provides such options only if its corresponding `VERIFY_SYNC_BOName` property evaluates to `verify` or `sync`.

The collaborations provided with `InterChange Server` have two configuration properties that provide options when a referenced business object in an array fails verification or synchronization. One of these properties handles failed `Contact` business objects; the other handles failed `Item` business objects.

The following example demonstrates how an array of `Contact` business objects is handled. In this example, both `SalesOrderProcessing` and `InstalledProductSync` delegate arrays of reference-valued `Contact` business objects. Both use the `FAIL_ON_CONTACT_ERROR` property to determine the collaboration's behavior when the called collaboration fails to verify or synchronize one of the `Contact` objects in the array.

If you modify `CollaborationFoundation` to delegate an array of `Contact` objects, maintain consistency with other collaborations by coding your collaboration to evaluate `FAIL_ON_CONTACT_ERROR` only if `VERIFY_SYNC_CONTACT`

evaluates to verify or sync and if ContactWrapper fails to verify or synchronize any of the referenced Contact objects. The product-delivered collaborations perform the following operations based on the value of FAIL_ON_CONTACT_ERROR:

- True—The collaboration stops processing.
- False—The collaboration continues processing and removes each failed Contact from the array of child business objects.

By default, the FAIL_ON_CONTACT_ERROR property is set to true.

The following paragraphs demonstrate how arrays of Item business objects are handled. WebSphere InterChange Server business objects can contain an array of references to Item and Contact business objects. Therefore, each generic Order can reference an array of generic Item objects; you can configure SalesOrderProcessing to delegate an array of referenced Item business objects to ItemWrapper.

To handle errors for an individual Item that fails verification or synchronization, SalesOrderProcessing uses the FIND_ALL_ITEM_ERRORS property. The collaboration evaluates FIND_ALL_ITEM_ERRORS only if VERIFY_SYNC_ITEM evaluates to verify or sync and if ItemWrapper fails to verify or synchronize any of the referenced Item business objects.

Figure 10 illustrates the way in which CollaborationFoundation handles an array of referenced Item business objects when VERIFY_SYNC_ITEM evaluates to sync and FIND_ALL_ITEM_ERRORS evaluates to true.

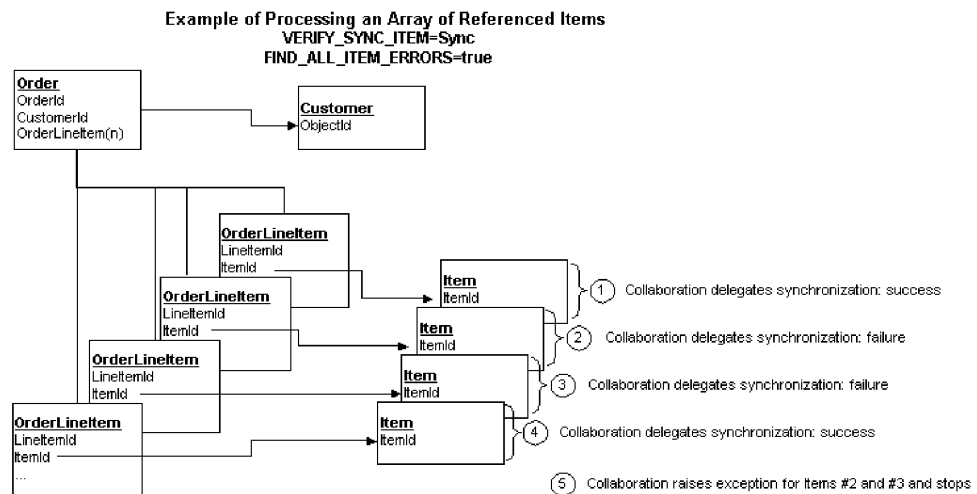


Figure 10. Example of processing an array of referenced Item business objects

Collaborations perform the following operations based on the value of FIND_ALL_ITEM_ERRORS:

- True—The collaboration continues processing and sending referenced Item business objects. After handling the last Item in the array, the collaboration raises an exception for each failed Item and stops processing. This setting enables an administrator to receive all item errors at once.
- False—The collaboration stops processing. The setting requires an administrator to restart the collaboration after each item error. The administrator must find and fix each error.

By default, the value of FIND_ALL_ITEM_ERRORS is false.

Processing multiple types of items: WebSphere InterChange Server delivers the generic Item business objects listed in Table 15.

Table 15. Generic Item business objects

Generic Item business object	Description
ItemBasic	Contains data attributes that are common across all the logical organizations that use item data. It is considered a prerequisite for other InterChange Server generic Item business objects.
ItemOrder	Contains data attributes that capture the set of fields maintained for a specific order management organizational entity.
ItemPlanning	Contains data attributes that pertain to planning for an item's future requirements and restocking.
Item	A flat business object that represents item data. Several collaborations use this business object as a placeholder. Generic Item enables collaborations to determine at runtime the actual type of Item business object to be processed. Therefore, generic Item <i>must</i> exist on an installation in order to bind a connector to a collaboration that processes generic ItemBasic, ItemOrder, and ItemPlanning.

If a triggering business object can reference an Item business object, it is important to allow the person configuring the collaboration object to specify which type of referenced Item to verify or synchronize. The ITEM_TYPE property provides this functionality.

If the collaboration you build from CollaborationFoundation has a triggering business object that references Item business objects, extend your collaboration to use the ITEM_TYPE property. Define its set of values to include every type of Item business object available on your system.

By default, the ITEM_TYPE property is set to Item.

Adding messages: To examine or modify CollaborationFoundation's messages, edit its message file (collaborations\messages\CollaborationFoundation.txt). Maintain consistency with other InterChange Server collaborations by creating messages that provide information if any aspect of the dependency checking fails.

Because the WebSphere InterChange Server synchronization and wrapper collaborations provide their own error messages, collaborations developed from CollaborationFoundation need not contain error messages specific to each failed synchronization or verification of related business objects. However, these customized collaborations must explicitly handle the messages received from their called collaborations. For more information on handling these messages, see "Raising exceptions" on page 37.

Adding Scenarios: To delegate checking of a referenced business object to another collaboration, you must create a collaboration scenario that evaluates whether the collaboration should verify or synchronize the referenced business object.

The scenario must check the value of the collaboration's VERIFY_SYNC_BOName property to determine whether the collaboration has been configured to verify or synchronize the referenced business object. You can also have the scenario perform other preliminary checking of the referenced business object before it calls a Wrapper collaboration.

For example, the following code enhances performance by preventing the collaboration from calling the CustomerWrapper collaboration unless it is truly required:

```
// Get the value of the configuration property
String vsc = getConfigProperty("VERIFY_SYNC_CUSTOMER");

// By default, do not branch to Wrapper call.
bBranch= false;

// If VERIFY_SYNC_CUSTOMER evaluates to "neither" or
// if the source application's business object does not contain the
// Customer's primary key, do not change the value of bBranch.
if (vsc.equals("neither") || (processingBusObj.isNull("CustomerID"))"
    {}
else
    {
    // Get the Customer's primary key from the source
    // and destination application's business objects.
    // Note: The DestinationAppRetrieveBusObj variable
    // contains a value only if the USE_RETRIEVE property evaluates
    // to "true" and has already been processed.
    String sc=processingBusObj.getString("CustomerID");
    String dc=DestinationAppRetrieveBusObj.getString("CustomerID");

    // If the Customer's primary key is the same in both the
    // source and destination application (therefore, no need to
    // synchronize), the verb is Update, and USE_RETRIEVE evaluates
    // to "true", do not change the value of bBranch.
    if (sVerb.equals("update") && bUseRetrieve && sc.equals(dc))
        {}
    else
        {
        // If the verb is Create, or if the Customer's primary
        // keys are not identical and the verb is Update, call
        // the Wrapper collaboration.
        bBranch = true;
        }
    }
}
```

For an example of a collaboration template that includes the above code, see SalesOrderProcessing. Its Additional Processing 3 subdiagram includes code that checks the values of its VERIFY_SYNC_CUSTOMER, VERIFY_SYNC_ITEM, and VERIFY_SYNC_CONTACT properties to determine which of several paths to follow.

Because SalesOrderProcessing processes the USE_RETRIEVE diagram before processing the Additional Processing 3 subdiagram, its DestinationAppRetrieveBusObj variable can already contain the destination application's business object.

Performing recursive checking of dependent data

Most business objects reference other business objects whose types are different from their own. For example, Order references Customer, Contact, and Item. Order does not reference another Order business object. Because Order is not the only generic business object that can reference Customer, Contact, and Item objects, the collaboration that modifies Order needs to delegate processing of the referenced business objects to the appropriate wrapper collaborations. To maintain data consistency in environments that run multiple collaborations, no two product-delivered collaborations modify a business object of the same type.

The discussion of business object modification becomes more complicated, however, when analyzing the collaborations that modify InstalledProduct and Item business objects.

InstalledProduct business objects and their parent InstalledProduct: Like Order, an InstalledProduct business object can reference Customer, Contact, and Item. Unlike any other generic business object, however, InstalledProduct can also reference another InstalledProduct as its parent. Figure 11 illustrates the relationship between multiple InstalledProduct business objects.

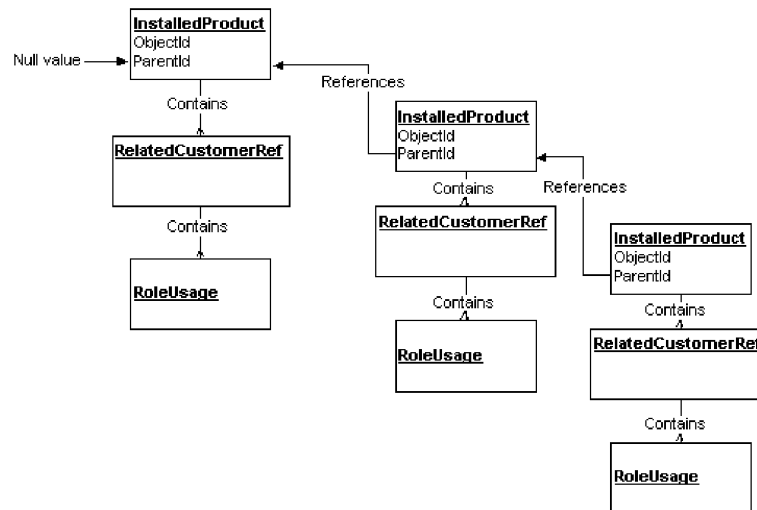


Figure 11. Relationship between multiple InstalledProduct business objects

Figure 11 shows that the parent InstalledProduct does not contain its child InstalledProduct. Instead, the child contains a reference to its parent in its ParentId attribute, whose type is String. Any InstalledProduct that has a null in its ParentId attribute is at the top of its product hierarchy.

Item business objects and their prerequisite Item objects: Item business objects also introduce a new level of complexity to business object modification. As shown in Table 15 on page 43, WebSphere InterChange Server delivers a family of Item business objects. The business integration system handles Item business objects flexibly; collaborations use the value of their ITEM_TYPE property to determine which type of Item they are configured to handle.

Although one Item does not reference another as its parent, an Item business object can reference another business object in the Item family as its prerequisite. For example, ItemOrder and ItemPlanning can both reference ItemBasic as a prerequisite. By default, the PREQ_ITEMORDER and PREQ_ITEMPLANNING properties of the ItemSync collaboration specify ItemBasic as the prerequisite for both of these Item business objects.

Recursively processing business objects of the same type or type family: When a business object references another business object of the same type or type family, the same collaboration that modifies the triggering business object can modify the referenced business object. In other words, the InstalledProductSync collaboration must delegate the handling of its triggering business object's referenced Customer, Contact, and Item objects.

However, InstalledProductSync can synchronize not only its triggering InstalledProduct but the parent InstalledProduct of the triggering business object, and the parent of the parent. Moreover, ItemSync can synchronize not only its triggering ItemOrder, but the ItemBasic prerequisite of the triggering business object. If the triggering business object's prerequisite has its own prerequisite, ItemSync can synchronize them all.

WebSphere InterChange Server provides two collaborations that recursively process a stack of business objects:

- InstalledProductSync—Builds a stack of parent InstalledProduct business objects by starting at the bottom with the triggering business object and building upward to the topmost parent. The collaboration then synchronizes the InstalledProduct business objects, moving from the top of the stack to the bottom. At each node in the hierarchy, the collaboration delegates handling of all referenced business objects that are not of the type InstalledProduct *before* it actually synchronizes the InstalledProduct business object. In other words, InstalledProductSync can recursively synchronize multiple business objects by using the Last-In-First-Out (LIFO) method.

For example, assume that an organization that builds cars manages the components as separate installed products within a product hierarchy. In this case, the fuel injector is handled as an installed product, the parent of the fuel injector (the engine) is handled as an installed product, and the parent of the engine (the car) is handled as an installed product. To synchronize the fuel injector, InstalledProductSync builds a stack; the fuel injector is located at the bottom of the stack. The collaboration places the car at the top of the stack. When synchronizing, InstalledProductSync synchronizes the car first and the fuel injector last.

InstalledProductSync is based on the assumption that the parent of an InstalledProduct object must exist *before* the InstalledProduct is created, much as a boss must exist before an employee is hired.

- ItemSync—Works its way down a hierarchy of prerequisite Item business objects, removing a business object from the hierarchy if its prerequisite business object already exists in the destination application. In other words, ItemSync performs a depth-first search when recursively synchronizing prerequisite business objects within the hierarchy.

ItemSync's process differs from the InstalledProductSync process as follows:

- ItemSync can handle any number of dependencies. The collaboration is designed to facilitate processing of generic business objects developed at a user's installation. With moderate modifications, ItemSync can accept user-defined business objects as triggering business objects or as prerequisites.
- As it moves down through its list of prerequisites, ItemSync evaluates each failed prerequisite to determine whether the prerequisite has a prerequisite above it in the hierarchy that was successfully retrieved. If it does, ItemSync keeps only the failed prerequisite on the list.

If you plan to modify CollaborationFoundation to perform recursive processing, examine the code for the InstalledProductSync and ItemSync collaborations. You can copy scenarios from those collaborations into the collaboration that you build from CollaborationFoundation. You can then modify the scenarios to meet your specific needs. For more information about these collaborations, see the reference pages for InstalledProductSync and ItemSync.

WrapperFoundation template

The WrapperFoundation collaboration template is a tool that makes it easy to develop user-defined wrapper collaborations that adhere to InterChange Server standards. A wrapper collaboration is a collaboration that handles the verification or synchronization of a business object for another collaboration. The calling collaboration sends a top-level business object that is referenced on its own triggering business object to the wrapper collaboration.

For example, the SalesOrderProcessing collaboration synchronizes the generic Order business object. Generic Order contains references to a generic Customer business object, which represents the customer making the order. Moreover, generic Order contains an array of generic OrderLineItem business objects. Each OrderLineItem references a generic Item business object, which represents the items ordered.

To modularize collaboration logic in such a situation, InterChange Server provides separate collaboration templates to process generic Order and the generic business objects that it references. For instance, InterChange Server provides the following templates to process an Order business object that references Customer and Item business objects:

- SalesOrderProcessing—Processes the order.
- CustomerWrapper and CustomerSync—Processes the referenced customer.
- ItemWrapper and ItemSync—Processes the referenced items.

Separating business object processing into different, specific collaborations not only enhances the reusability of each collaboration template, but also prevents two collaborations from modifying the same data at the same time. For more information, see “Problems in concurrent processing” on page 55.

To maintain consistency across all synchronization and access collaborations, WebSphere InterChange Server builds the templates for these collaborations from the CollaborationFoundation template. To maintain consistency across all wrapper collaborations, all wrapper collaboration templates are built from the WrapperFoundation template. You can use WrapperFoundation to develop your own wrapper collaborations.

WrapperFoundation features

A wrapper collaboration object generated from the WrapperFoundation template can verify the existence of its triggering business object in the destination application or facilitate its synchronization. This section contains the following topics:

- “Verification processing”
- “Synchronization processing” on page 48

Verification processing

When a wrapper collaboration receives its reference-valued triggering business object with the Exists verb, it verifies the existence of its triggering business object by retrieving it from the destination application. The wrapper collaboration uses the Retrieve verb for this verification.

If the verification succeeds, the wrapper collaboration returns a success status to its calling collaboration.

If the verification fails, the wrapper collaboration's behavior depends on the setting of its `CONTINUE_WITH_WARNING` configuration property.

For more information and an illustration of the verification process, see the user guide for the particular wrapper collaboration you are using.

Synchronization processing

When a wrapper collaboration receives its reference-valued triggering business object with the Sync verb, it facilitates the synchronization of the business object. It does so by retrieving all values from the source application and sending the full-valued business object with the Create verb to the appropriate synchronization collaboration.

If the synchronization collaboration succeeds in creating the business object in the destination application, the wrapper collaboration returns a success status to its calling collaboration.

If the synchronization collaboration fails to create the business object in the destination application, the wrapper collaboration's behavior depends on the setting of its `CONTINUE_WITH_WARNING` configuration property.

To prevent the called synchronization collaboration from attempting to create an existing object and returning a failure status to the wrapper collaboration, use the following settings for the synchronization collaboration:

- Set the `USE_RETRIEVE` configuration property to true.
- Set the `INFORMATIONAL_EXCEPTIONS` configuration property to 3010.

For more information and an illustration of the synchronization process, see the user guide for the wrapper collaboration template you are using.

Using the WrapperFoundation template

Perform the following tasks when you want to use the WrapperFoundation template:

1. Copy the entire WrapperFoundation template to your development area.
2. Rename the template to reflect the collaboration you are going to build.
3. From within Process Designer, use the Ports and Triggering Events tab of the Template Definitions window to change the type of existing ports (DestinationAppRetrieve, From, SourceApp, and To) to assign the appropriate business objects to them. For example, to create an InvoiceWrapper template, you must change each port's business object from the default value (Controller) to the Invoice business object. For more information, see "WrapperFoundation ports."

WrapperFoundation ports

Figure 12 on page 49 illustrates WrapperFoundation's ports. The tables that follow the figure provide information about each port.

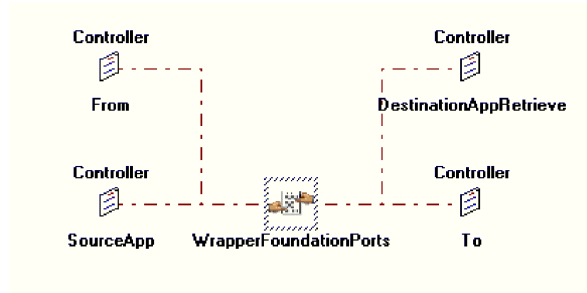


Figure 12. WrapperFoundation collaboration ports

WrapperFoundation’s DestinationAppRetrieve port

Table 16 lists the features of the WrapperFoundation template’s DestinationAppRetrieve port.

Table 16. Port features for the WrapperFoundation DestinationAppRetrieve port

Port feature	Value
Business Object	Business object for which the wrapper collaboration is named. For example, CustomerWrapper sends a reference-valued Customer business object. This business object defaults to Controller.
Verbs Used	Retrieve
Purpose	Retrieves the business object from the destination application
Bound To	Destination application’s connector

WrapperFoundation’s From port

Table 17 lists the features of the WrapperFoundation template’s From port.

Table 17. Port features for the WrapperFoundation From port

Port feature	Value
Business Object	Business object for which the wrapper collaboration is named. For example, CustomerWrapper is triggered by a Customer business object. The default value is Controller.
Verbs Used	Sync, Exists
Purpose	Receives the triggering business object from the calling collaboration
Bound To	The ToBusObjWrapper port of the calling collaboration

WrapperFoundation’s SourceApp port

Table 18 lists the features of the WrapperFoundation template’s SourceApp port.

Table 18. Port features for the WrapperFoundation SourceApp port

Port feature	Value
Business Object	Business object for which the wrapper collaboration is named. For example, CustomerWrapper retrieves a Customer business object from the source application. The business object defaults to Controller.
Verbs Used	Retrieve
Purpose	Retrieves the triggering business object from the source application.
Bound To	Source application’s connector

WrapperFoundation's To port

Table 19 lists the features of the WrapperFoundation template's To port.

Table 19. Port features for the WrapperFoundation To port

Port feature	Value
Business Object	Business object for which the wrapper collaboration is named. For example, CustomerWrapper sends a reference-valued Customer business object. This business object defaults to Controller.
Verbs Used	Create
Purpose	Sends a business object out of the collaboration. Usually used to send a full-valued business object to the relevant synchronization collaboration,
Bound To	The From port of the collaboration that synchronizes or subscribes to the business object

Extending WrapperFoundation

CustomerPartnerWrapper and ItemWrapper provide examples of WebSphere InterChange Server collaborations that were built from WrapperFoundation and were modified to meet specific needs. This section describes those modifications as examples of ways you can modify the WrapperFoundation template.

CustomerPartnerWrapper

Because different applications use CustomerPartner data in significantly different ways, CustomerPartnerWrapper handles its business object's keys in a unique way. Some applications require CustomerPartner to contain the identifier of its associated Customer object as well as its own identifier. Other applications do not require the Customer object's identifier and do not provide that identifier when sending the business object to the InterChange Server business integration system.

If a collaboration receives a CustomerPartner business object from a system that does not provide the Customer identifier, but must synchronize the object to a system that does require the identifier, the collaboration must obtain the additional data and provide it to the destination application. For this reason, CustomerPartnerWrapper does *not* use the command provided by WrapperFoundation to set the business object's keys:

```
SourceAppBusObj.setKeys(triggeringBusObj);
```

Instead of using the above command, which WrapperFoundation provides in the Sync scenario's Retrieve from Source node, CustomerPartnerWrapper gets the values for the key in the following two separate statements:

```
SourceAppBusObj.set("ObjectId",  
    triggeringBusObj.getString("ObjectID"));  
SourceAppBusObj.set("AdditionalKey",  
    triggeringBusObj.getString("AdditionalKey"));
```

Also located in the Sync scenario's Retrieve from Source node, these statements use both the ObjectId and the AdditionalKey attributes as the potential key.

In the Sync scenario's Prepare Object for Sync Collaboration node, WrapperFoundation contains the following code:

```
ToBusObj =(SourceAppBusObj).duplicate();  
ToBusObj.setKeys(triggeringBusObj);
```

In the equivalent node, CustomerPartnerWrapper uses two separate statements to set the key for ToBusObj:

```
ToBusObj =(SourceAppBusObj).duplicate();
ToBusObj.set("ObjectId", triggeringBusObj.getString("ObjectId"));
ToBusObj.set("AdditionalKey", triggeringBusObj.getString("AdditionalKey"));
```

ItemWrapper

Table 15 on page 43 lists the four different types of generic item business objects delivered with WebSphere InterChange Server. It is assumed that you are planning to define your own types of generic item business objects to meet the specific needs of your environment. Therefore, the ItemSync collaboration was designed to be easily extended to handle user-defined triggering and prerequisite item business objects. For more information, refer to the user's guide for the ItemSync collaboration.

Because ItemWrapper can be triggered by any type of generic Item business object, the collaboration must first determine the type of business object that triggered it. Therefore, ItemWrapper's Scenario Definitions for both the Verify and Sync scenarios contain the following statement, which is not found in other wrapper collaboration templates:

```
BusObj getItemFlvrBusObj = new BusObj(triggeringBusObj.getType());
```

The above statement creates a business object unique to ItemWrapper (getItemFlvrBusObj). This statement also gets the type of the triggering business object and uses it to set the type of the business object used in ItemWrapper's verification and synchronization scenarios.

All wrapper collaborations create a DestinationAppBusObj and a SourceAppBusObj business object in the Declarations section of the collaboration's Template Definition. The type of these business objects is the same as that of the triggering business object. Whereas other wrapper collaborations use DestinationAppBusObj and SourceAppBusObj to retrieve data from the destination and source applications, ItemWrapper uses getItemFlvrBusObj. Before retrieving the item from the destination application in the Verify scenario or from the source application in the Sync scenario, ItemWrapper uses the key values of triggeringBusObj to set the keys of getItemFlvrBusObj, as follows:

```
getItemFlvrBusObj.setKeys(triggeringBusObj);
```

In the Sync scenario, ItemWrapper uses the values in getItemFlvrBusObj to create the business object that it sends to the destination, as follows:

```
toItemFlvrBusObj = getItemFlvrBusObj.duplicate();
//toItemFlvrRef.copy(getItemFlvrRef);
toItemFlvrBusObj.setKeys(triggeringBusObj);
```

A standard wrapper collaboration, such as ContactWrapper, creates a ToBusObj, which it instantiates from SourceAppBusObj. The following example illustrates this creation and instantiation:

```
ToBusObj =(SourceAppBusObj).duplicate();
ToBusObj.setKeys(triggeringBusObj);
```

Because ItemWrapper does not use the standard FromBusObj, the collaboration's code comments out the code in the Initialize Variables node of both the Verify and Sync scenarios.

Other extensions to the template

You can extend WrapperFoundation to contain additional ports, properties, processing logic, and messages. For more information, see “Extending CollaborationFoundation” on page 35.

Building collaboration groups

A *collaboration group* is a set of collaboration objects that represents a combined business process. A collaboration group lets you combine discrete units of logic. The collaboration objects are bound to each other through the same types of ports through which they can also bind to connectors.

Collaboration groups provide the following benefits:

- Enable you to modularize logic. You can develop and test a unit of logic once, and then deploy it multiple times.
- Enable you to expand existing collaborations. You can create collaboration templates that call or are called by existing collaborations.

Collaboration groups are formed from two or more collaborations. Within a group, collaborations are bound to other collaborations, and there is always the notion of a caller collaboration and a called collaboration. For any two collaborations that are bound to each other, one is the caller collaboration and one is the called collaboration. A caller collaboration is bound such that one of its service calls sends a business object that triggers the execution of another collaboration. The called collaboration receives the business object, which is its triggering event. The called collaboration returns the result to the caller after executing. See Figure 13.

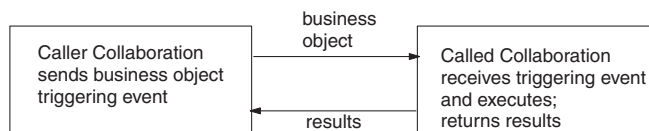


Figure 13. Caller and called collaborations

Within a collaboration group, a collaboration that does not support long-lived business processes cannot bind to a collaboration that is deployed as a long-lived business process.

Example of a collaboration group: Customer Manager

An example of a collaboration group is the WebSphere InterChange Server product Customer Manager, which consists of the following collaborations:

- CustomerSync
- CustomerWrapper
- CustomerPartnerSync
- CustomerPartnerWrapper

When you install the Customer Manager collaboration, you receive all collaboration templates. You can configure them and bind them (establish the communication between collaborations using ports) in various ways to form a unified process.

The CustomerSync collaboration synchronizes a SoldTo customer; that is, the CustomerSync collaboration ties together events and data with the SoldTo

customer. You can also choose to synchronize data and events about related customer information. In that case, you could bind CustomerSync to the CustomerPartnerWrapper, which performs some preprocessing, and then bind CustomerPartnerWrapper to CustomerPartnerSync. Figure 14 illustrates this set of bindings.

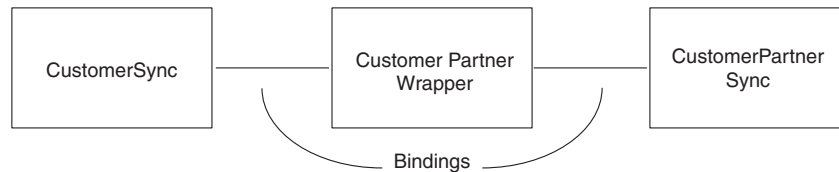


Figure 14. Bound collaboration group

Creating a collaboration group

Here are the general steps for creating a collaboration group:

- In the calling collaboration:
 - Create a port for the type of business object to be passed to the called collaboration.
 - Set up a service call that passes the business object along with the verb that you want the called collaboration to handle.
 - Handle the results of the service call as usual.

Note: If one of the collaborations in a collaboration group is configured for Service Call In-Transit persistence, all collaborations in that group will be configured automatically by the WebSphere InterChange Server system to maintain consistent recovery behavior. For more information, see “Service calls and exactly-once requests” on page 178.

- In the called collaboration:
 - Create a port for the type of business object to be received from the caller collaboration.
 - Create a scenario and assign the triggering event to the scenario.

Including Web Services

Business Integration Server supports the use of Web services in collaborations. A Web service is a modular application whose public interfaces and bindings are defined with XML, and it is accessible through open protocols such as HTTP and SOAP. A Web service can be included in a collaboration template’s activity definition. When the corresponding collaboration object is run, the Web service is automatically invoked. No modification of InterChange Server is required.

Use System Manager to locate and register Web services. Registered Web services become part of the Integration Component Library (ICL) project. In addition, any business objects needed for the Web services are automatically generated and placed in the ICL project. For more information on using System Manager to register and manage Web services, see the System Implementation Guide.

To include a Web service in a collaboration template, you must export it from the ICL project in System Manager. Each method is exported as a function block to Activity Editor, where it can be placed in an activity definition. For more information on exporting a Web service and adding it to an activity definition, see “Web services function blocks” on page 155.

Designing for long-lived business processes

If you plan to deploy your collaboration as a long-lived business process, keep the following in mind when designing and building the collaboration template:

- Use global template or port variables for any data that you want to persist through the business process.
- The references for all `CwDBConnection` objects are released before a service call in a long-lived business process environment, and all active database transactions are implicitly committed. If necessary, design your template to re-acquire the `CwDBConnection` objects after the service call has finished. In addition, reinitialize the database transaction context after the service call if you are using explicit database transaction bracketing.
- If the collaboration is going to be bound to an adapter, ensure that the adapter is configured to use JMS as the transport mechanism. Long-lived business processes cannot use an adapter with any other type of transport.
- Long-lived business process collaborations cannot be bound to external Access Clients.
- Within a collaboration group, collaborations that do not support long-lived business processes cannot bind to a long-lived business process collaboration.

Dynamic service calls

There are two approaches a collaboration can use to request services in the WebSphere InterChange Server. The first method is through a static call; you define the destination collaboration or connector for the service call when you design the collaboration and bind it to a specific port. This makes the service call static in that it can only communicate with the explicitly bound destination.

The second method is by making a dynamic service call to the destination collaboration or connector. This method allows you to dispatch business objects to multiple destinations that are not known when you are designing the collaboration. You implement a dynamic service call using `dynamicSend()`.

The advantages of using dynamic service calls are:

- There is no requirement to bind a port at design time. The information needed to process the business object is part of the service call.
- The collaboration can interact with many connectors providing greater flexibility in a dynamic processing environment.
- The collaborations handle the requests synchronously.

When deciding whether to use dynamic service calls, consider the following:

- You must understand event sequencing when using dynamic service calls. Because there is no port explicitly configured with a dynamic service call, the dynamic service call may change the event sequencing instance queue.
- Dynamic service calls cannot support compensation. There is no rollback method available when using a dynamic service call.
- InterChange Server does not consider a dynamic service call to another collaboration as establishing a collaboration group. This is because there is no explicit port binding between the two collaborations.
- A long-lived business process cannot be a consumer of a dynamic service call.
- A long-lived business process can make a dynamic service call to a connector but the connector must use JMS as its transport type.

Designing for parallel execution

InterChange Server provides a parallel-execution environment: it can run multiple collaborations concurrently in separate threads and it can also run multiple threads of the same collaboration (known as multithreading).

Attention: The thread pool for collaborations is used *only* for event-triggered flows and not for call-triggered flows. However, call-triggered flows are also multi-threaded in execution in that they use the thread pool of the IBM Java Object Request Broker (ORB).

Multithreading capabilities

Each server has a maximum number of threads that can be simultaneously spawned to process business object subscriptions. You can set your own maximum number of threads to be spawned, based on your individual situation and what you determine to be optimal for performance. Of course, the number you set cannot be greater than the number of threads allowed by the server.

To set the maximum number of threads that can potentially be spawned for a single collaboration object, specify the maximum number in the **Maximum number of concurrent events** field, which System Manager displays in the Collaboration General Properties tab for the collaboration object. You can access this tab from the Properties option of the context menu of a collaboration object, either from within a project or from within the Server View.

Note: If the destination connector is configured for parallel processing, code the collaboration template to verify that the request was successfully sent to the application. Add this code to the node immediately following the exception transition link for the service call. For more information, see `getSubType()` in Chapter 29, “CollaborationException class,” on page 423.

Important: If the destination connector is single-threaded, it must be configured for parallel processing to take advantage of a multi-threaded collaboration. For more information, see the *Implementation Guide for WebSphere InterChange Server*.

Problems in concurrent processing

In any concurrent processing environment, there is always the danger of data inconsistency. Data inconsistency can occur whether the concurrent processing is by means of multiple processes or multiple threads. If two programs or two threads access the same data at the same time, there is always the possibility that one may modify the data and adversely affect the operations of the other program or thread in unexpected ways. Concurrent processing environments handle this problem by synchronizing access to shared data; a thread or process locks a portion of data so that another thread or process cannot simultaneously access it.

For a simple example of the problem as it might occur in the business integration environment, consider the following situation:

- An application user at a WebSphere InterChange Server source application must add \$10,000 to an employee’s \$40,000 salary.
- The application user accidentally enters a salary increase of \$100,000. The user realizes that the entry was incorrect and updates the salary again, this time subtracting \$100,000.

- Both operations result in the sending of an Employee.Update event for the same employee ID to the WebSphere InterChange Server system.
- The events are processed out of order and sent to another application for synchronization.
- In the destination application, the first update operation attempts to set the employee's salary to -\$60,000—it tries to subtract \$100,000 from the employee's salary of \$40,000. This causes a semantic error and unexpected results, because employee salaries cannot fall below zero. The second update could then encounter an error also.

InterChange Server has the following features to ensure data consistency and address this problem:

- "Event sequencing"
- "Event isolation"

Event sequencing

Event sequencing ensures that two threads of the same collaboration do not work on the same data concurrently. If multiple events have the same business object type and key values, the server queues them and delivers them in order of arrival. The collaboration thread that receives the first event must complete before the collaboration receives the next event. Event sequencing thereby preserves execution order, even in the presence of multi-threaded execution, despite that the various threads could execute at varying speeds.

You do not need to design a collaboration in any special way to take advantage of event sequencing; it is done automatically.

Note: You can turn off event sequencing by changing the value of the connector's **ControllerEventSequencing** property to false and unchecking the **Event Isolation** box in the collaboration configuration. See "*Configuring connectors*" in *IBM WebSphere Interchange Server: Implementation Guide for WebSphere Interchange Server*.

Event isolation

Event isolation ensures that two collaborations do not work on the same data concurrently. Sometimes multiple collaborations handle the same types of business objects. An event arrives and triggers a particular collaboration. This collaboration starts its execution and, while it executes, it has sole access to that business object instance in InterChange Server. If another event relating to the same data arrives, InterChange Server queues the newly arrived event until the executing collaboration completes its processing of the first event. Some restrictions apply to this feature; they are described in the following sections.

InterChange Server does not do event isolation automatically. Collaboration developers must design templates in a certain way to take advantage of event isolation. This section describes the rules and gives some examples of design decisions that help achieve this goal.

Note: These guidelines apply only to collaborations that perform operations that change data and operate in an environment where multiple collaborations are in use. If you are developing a collaboration that performs only retrieve operations and will always be the sole collaboration using that business object type on its server, you can disregard these guidelines.

When event isolation is applied

InterChange Server determines the application of event isolation at runtime, based on an analysis of the events that arrive and the ports of active collaborations. The criteria for event analysis is the same as for event sequencing: events are the same when the business object type and key values are the same.

InterChange Server also uses the configured value of the collaboration's **Event Isolation** property to determine whether to enforce event isolation. See "*Configuring collaboration objects*" in *IBM WebSphere InterChange Server: Implementation Guide for WebSphere InterChange Server*.

Important: If you uncheck the **Event Isolation** property for any collaboration that needs the isolation, the potential for erroneous updates exists. For example, if collaboration A, B and C all reference the same business object and have the same destination, they should all have event isolation enabled. If any one of the collaborations has the property turned off, that collaboration may access a business object at the same time as another collaboration is updating the values in the object.

The analysis of active collaborations considers the set of each collaboration's ports that are bound to connectors. In *port matching*, InterChange Server checks whether:

- Among any of the collaborations, the ports are bound to the same set of connectors
- Among the ports bound to the same set of connectors, the ports bound to the same connector have the same business object type

For example, two collaborations have matching ports if both have these port bindings:

Connector1/Business object type A

Connector2/Business object type B

It is not important whether a port is used for incoming events or outgoing requests and responses; only the connector binding and the business object type matters.

Ports bound to other collaborations are not considered when determining the collaborations for which event isolation applies.

Port matching example: ports that match: Figure 15 illustrates two collaborations, X and Y, for which event isolation would apply. The small black rectangles at the edges of the collaborations indicate ports.

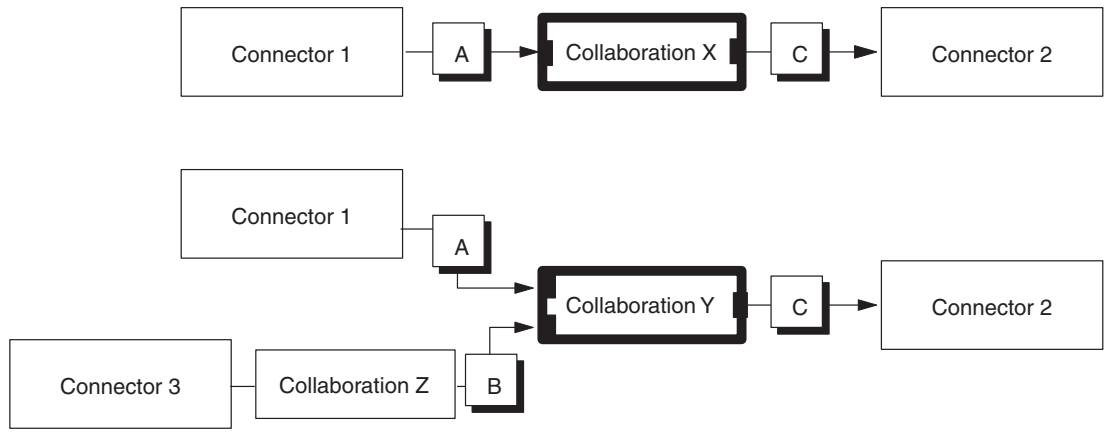


Figure 15. Matching ports

In Figure 15, X has two ports and Y has three ports. However, port matching considers only the two ports of Y that are bound to connectors; it disregards the port that is bound to collaboration Z. Both collaborations have the following ports bound to connectors:

- One port defined for business object type A and bound to connector 1.
- One port defined for business object C and bound to connector 2.

This example meets the criteria for event isolation, and the server isolates the incoming or triggering event. Therefore, event A instances would be subject to isolation in these two collaborations.

Port matching example: ports that do not match: Keep in mind that the server considers all ports when comparing collaborations; it does not confine port matching analysis to ports that receive triggering events. If two collaborations receive the same type of event from the same connector but send an outgoing business object to two different connectors, their events are not isolated.

Figure 16 illustrates two collaborations whose outgoing ports are bound to different connectors. Their event instances are not isolated.

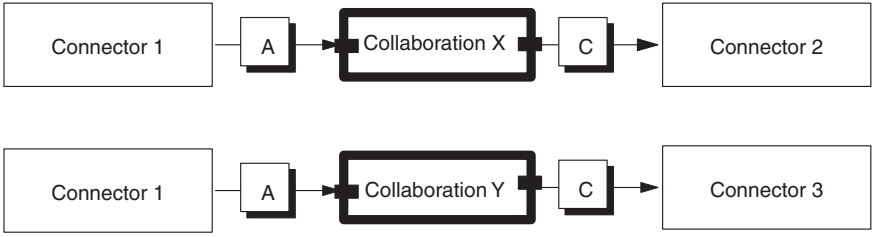


Figure 16. Unmatched ports

Design rules

You must design collaborations in a certain way if you want to benefit from event instance isolation. This section describes how to:

- Use delegation to form collaboration groups
- Handle child business objects as reference-valued objects

Using delegation: Each collaboration template that modifies a business object should be dedicated to modifying only that type of business object. If the

collaboration needs to modify another type of business object, such as a child business object, then you should create a separate collaboration whose purpose is to modify the other business object. Then, have the first collaboration *delegate* (pass) the other business object to the second collaboration for modification.

The rule of dedicating a single collaboration to modifying only one type of business object helps maintain data consistency. It prevents multiple collaborations from concurrently modifying the same instance of the same type of business object. Delegation ensures that the data consistency of a child business object is maintained with respect to instances of the same business object that are processed by other collaborations. Remember that you can use a business object in one context as a child and in another context on its own.

Imagine that you need to write a business process that deals with a business object A and a set of information, B, associated with it as a child business object. The business object structure might look similar to the illustration shown in Figure 17, where the B business object is a child of the A business object.

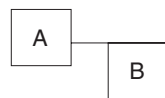


Figure 17. Example of a hierarchical business object

If a collaboration already exists that processes B business objects, you should delegate work on the B child business object to that collaboration. Alternatively, you might need to create another collaboration.

When you want to work on the data associated with the A business object, you must work on both the A business object and its set of B data, or the child business object. You would therefore create two different collaboration templates—one collaboration modifies business object A and the other modifies the child business object B—and you might combine these two templates into a collaboration group. Each collaboration template handles the operations on one business object.

Figure 18 illustrates a collaboration group, A/B, that contains an A-Processor collaboration and a B-Processor collaboration. The A-Processor collaboration processes the A business object. When the A-Processor collaboration needs to modify the B child business object, it uses a service call to send the B business object to the B-Processor collaboration. In Figure 18, a dotted line shows delegation.

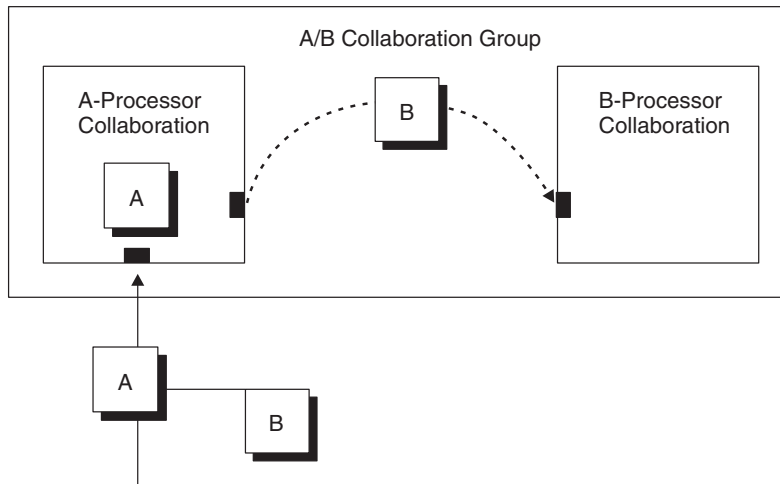


Figure 18. Delegation of a child business object

Handling child business objects as reference-valued: When a collaboration receives a delegated child business object (such as the case of the B-Processor collaboration in Figure 18), it should treat the business object as a reference-valued business object. A *reference-valued business object* contains only the values for attributes that are defined as primary keys for the business object. A *full-valued business object*, in contrast, contains values for other attributes.

In the figures in this chapter, reference-valued business objects are marked (r) and full-valued business objects are marked (f). Figure 19 is an example of a business object with a reference-valued child.

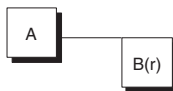


Figure 19. Hierarchical business object with a reference-valued child

Depending on the originating connector, events might be sent with reference-valued or full-valued child business objects. A collaboration that receives a delegated child business object might therefore receive all of its attribute values or only its primary key values. However, the collaboration should always treat the delegated child business object that it receives as reference-valued. It should assume only that the primary key values are correct; it should ignore attribute values that are not primary keys.

If the collaboration needs to perform operations on the child business object's non-key attributes, it must resolve the reference by retrieving the full-valued version of the business object from the source application. If the child business object is reference-valued, the retrieve operation obtains the additional attribute values. If the child business object is full-valued, the retrieve operation ensures that the associated data is current and valid.

Figure 18 illustrates the delegation of B as a reference-valued business object and the collaboration's resolution of the reference by retrieval from the source collaboration.

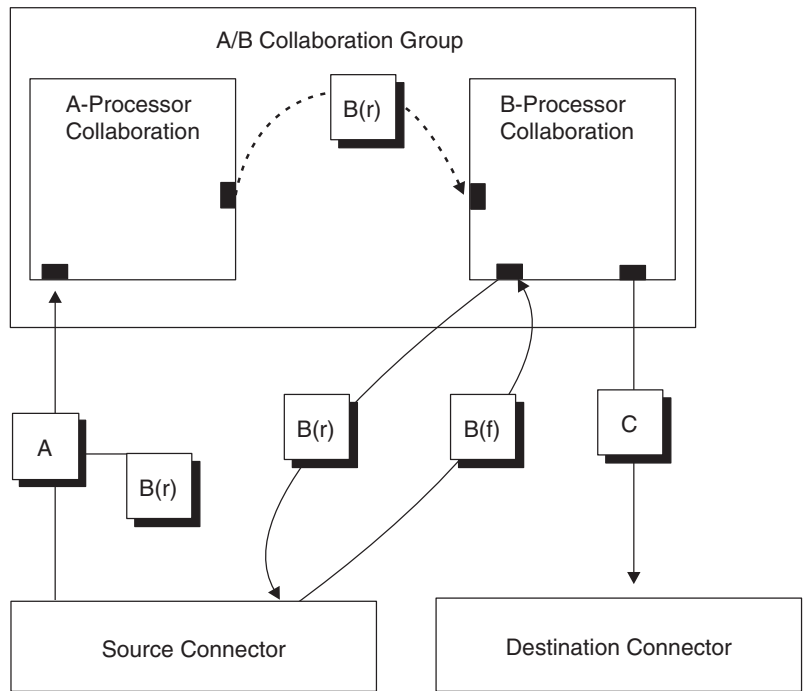


Figure 20. Resolving the reference

Examples

Figure 21 illustrates an environment in which event isolation is in effect between two different collaborations, B-Processor collaboration and B-to-C collaboration.

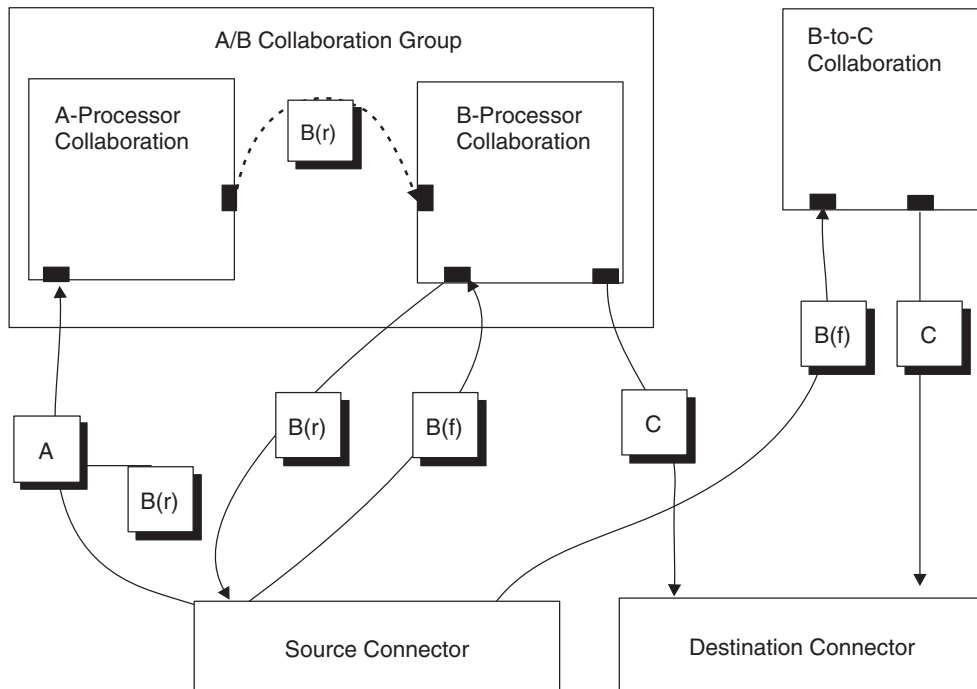


Figure 21. Two collaborations subject to event isolation

Notice that both the B-Processor collaboration and the B-to-C collaboration:

- receive events of business object type B
- produce business objects of type C
- are bound to the same set of connectors

Therefore, port matching would result in event isolation for these collaborations.

The next example (shown in Figure 22) illustrates how you can use collaboration objects created from the same template in two different ways in the same environment. This practice enables you to reuse and extend an existing collaboration template, such as when you want to add features to the collaboration.

Suppose that a Y-Processor collaboration template exists and that a collaboration object instantiated from the Y-Processor template, Y-Processor Collaboration1, is in use. You want to create new collaboration features that include and extend the functions of the Y-Processor collaboration template.

One way to do this is to reuse the Y-Processor collaboration template and create a new Y-Processor collaboration object that you use in a collaboration group. That is, you instantiate a second Y-Processor collaboration object, Y-Processor Collaboration2, from the Y-Processor template and place it in a collaboration group. There are now two Y-Processor collaborations for which event isolation is needed. An intermediary collaboration—Collaboration Z in the example—can provide additional functions and ensure event isolation, without requiring changes to Y-Processor.

In Figure 22, the server applies event isolation to the Y business objects received by the collaborations with dark outlines, Collaboration Z and Y-Processor Collaboration1. The numbers indicate the sequence of processing.

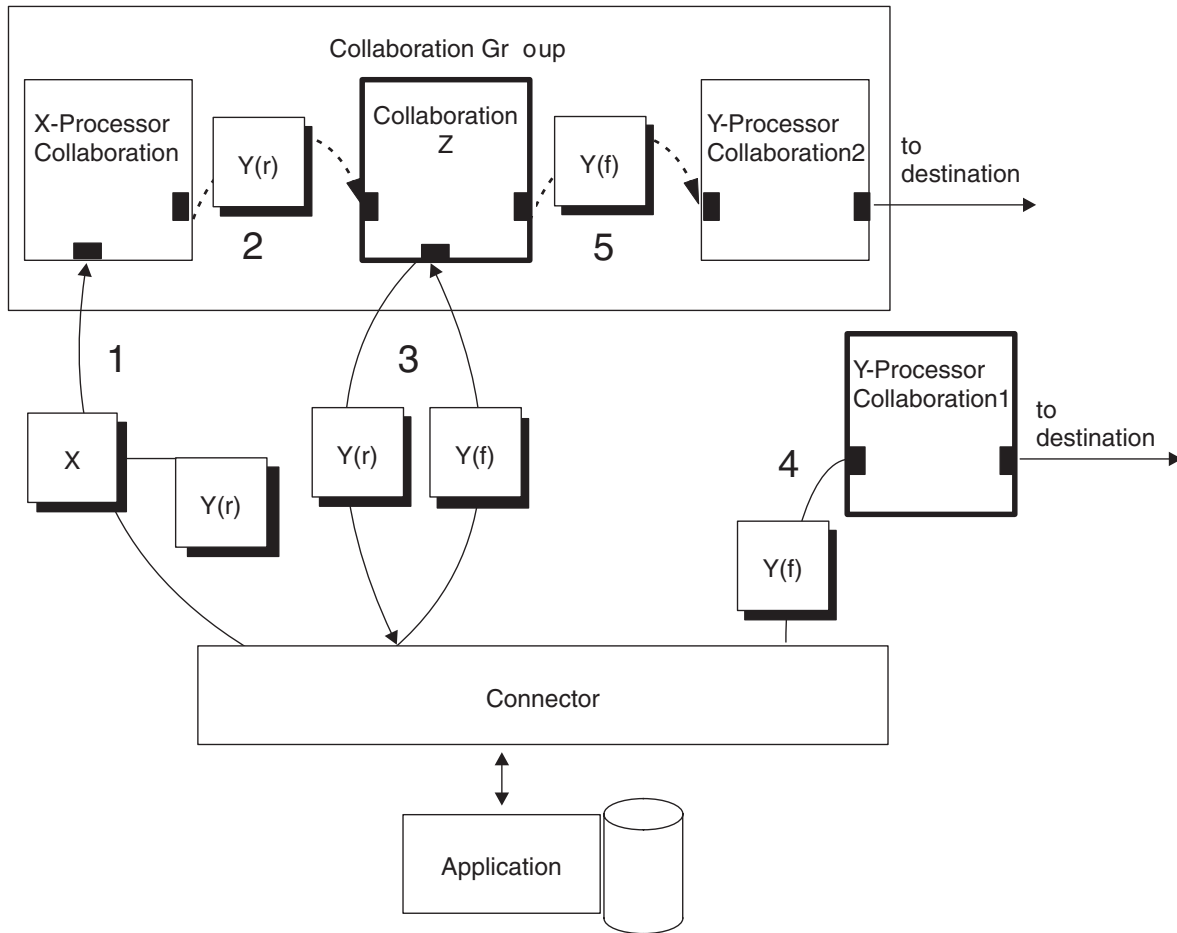


Figure 22. Retrieving a full-valued business object

Collaboration Z and Y-Processor Collaboration2 work as a team, in terms of event isolation. The guidelines for delegated business objects are followed by Collaboration Z on behalf of Y-Processor Collaboration2.

An internationalized collaboration

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

This section provides the following information on an internationalized collaboration:

- "What is a locale?" on page 64
- "Design considerations for an internationalized collaboration" on page 64

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 EBCDIC 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 collaboration

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

- “Locale-sensitive design principles”
- “Character-encoding design principles” on page 69

Locale-sensitive design principles

To be internationalized, a collaboration 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 collaboration should obtain its error messages from an English-language message file.

IBM WebSphere InterChange Server provides you with an internationalized version of InterChange Server and the collaboration runtime environment. It also provides you with CollaborationFoundation, a set of collaboration templates that facilitates the development of user-defined standards that adhere to WebSphere InterChange Server standards. (For more information on CollaborationFoundation, see “CollaborationFoundation template” on page 29.) CollaborationFoundation code is internationalized. Therefore, any parts of your collaboration template that use

CollaborationFoundation code will be internationalized. However, you must ensure that any custom code that you create is internationalized.

Note: The collaboration code that Process Designer creates is *not* internationalized. Once Process Designer generates your collaboration code, you must take the steps outlined in this section to internationalize your collaboration template.

Table 20 lists the locale-sensitive design principles that an internationalized collaboration must follow.

Table 20. Locale-sensitive design principles for collaborations

Design principle	For more information
The text of all error and status messages needs to be isolated from the collaboration template in a message file and translated into the language of the locale.	“Text strings”
The locale of a business object must be preserved during execution of the collaboration.	“Business object locales” on page 67
Properties of collaboration configuration properties must be handled to include possible inclusion of multibyte characters.	“Collaboration configuration properties” on page 68
Other locale-specific tasks must be considered.	“Other locale-sensitive tasks” on page 69

Text strings: It is good programming practice to design an internationalized collaboration so that it refers to an external message file when it needs to obtain text strings rather than hardcoding text strings in the collaboration code. When a collaboration needs to generate a text message, it retrieves the appropriate message by its message number from the message file. Once all messages are gathered in a single message file, this file can be localized by having the text translated into the appropriate language or languages. For more information on globalized message files, see Chapter 10, “Creating a message file,” on page 231.

To globalize the logging, exception, and email operations, make sure that all these operations use message files to generate text messages. By putting message strings in a message file, you assign a unique identifier to each message. Table 21 lists the types of operations that use a message file and the associated Collaboration API methods in the BaseCollaboration class that the collaboration template uses to retrieve their messages from a message file.

Table 21. Methods to retrieve messages from a message file

Message-file operation	BaseCollaboration method
Logging	logInfo(), logError(), logWarning()
Exception Handling	raiseException()
Email Notification	sendMail()

Note: InterChange Server standards recommend that trace messages are *not* included in a collaboration message file. Trace messages do not need to display in the language of the customer’s locale because they are intended for the product debugging process.

Handling logging and exception messages: To ensure that logging and exception-handling are always obtained from the collaboration message file, do *not* use the forms of the methods in Table 21 that allow you to specify the message string directly within the call. For example, to log an error to the log destination, do *not* use the following call to `logError()`:

```
logError("Log this message to the log destination");
```

Instead, create a unique identifier for the message and place the text within the collaboration message file. If this message were assigned a unique identifier of 712, its entry in the message file would appear as follows:

```
712  
Log this message to the log destination.
```

You can optionally add message parameters to this message as needed.

In an internationalized collaboration, the preceding call to `logError()` should be replaced with the following call, which obtains the log message from the collaboration message file:

```
logError(712);
```

Similarly, you should obtain all exception messages from the collaboration message file by avoiding use of the following form of `raiseException()`:

```
void raiseException(String exceptionType, String message)
```

Instead, use one of the `raiseException()` forms that includes a message number.

Handling email messages: The `sendEmail()` method allows you to send a message to specified email recipients. In an internationalized collaboration, email messages should go in the collaboration message file. However, the `sendEmail()` method does *not* provide a form that allows you to specify the unique identifier of a message. Therefore, to send an email message, you must first extract the message from the message file and then use `sendEmail()` to send the retrieved message string. Table 22 shows the method that the collaboration can use to retrieve a message from a message file.

Table 22. Method to retrieve a message from the message file

Collaboration library class	BaseCollaboration method
BaseCollaboration	getMessage()

The following code fragment retrieves message 100 from the collaboration message file and includes this message as part of an email message:

```
String retrievedMsg = getMessage(100);  
sendEmail(retrievedMsg, subjectLine, recipientList);
```

Handling miscellaneous strings: In addition to handling the message-file operations in Table 21, an internationalized collaboration template should *not* contain any miscellaneous hardcoded strings. You should isolate these strings into the message file as well.

To globalize hardcoded strings, take the following steps:

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

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

- In the collaboration template, use the `getMessage()` method to specify the isolated string by its message number.

For example, suppose your collaboration template contains the following line of code with a hardcoded string:

```
String imsg100 = "*****Before entering order-to-ATP map*****";
```

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

```
100
*****Before entering order-to-ATP map*****
[EXPL]
ATP Transaction: 162
```

In the collaboration template, replace the code that contains the hardcoded string with code that retrieves the isolated string (message 100) from the message file:

```
String imsg100 = getMessage(100);
//retrieve the message numbered ' 100'
String imsg100 = getMessage(100);
//display the retrieved message
```

For more information on the use of message files, see Chapter 10, “Creating a message file,” on page 231.

Business object locales: During execution of a collaboration object, there are two different locale settings:

- A collaboration inherits its locale, called a *collaboration locale*, from the InterChange Server instance in which the collaboration is running. The collaboration locale determines the locale of text messages that the collaboration uses for logging, tracing, exceptions, and email.
- A collaboration uses a *flow locale* for its triggering business objects. The flow locale determines the locale settings of the business objects that are involved while the collaboration executes.

When a business object is created, it always has a locale associated with its data. By default, every business object created in a collaboration uses the collaboration locale. However, a business object often needs to have the locale of the triggering business object (the flow locale). Because this collaboration locale might be different from the flow locale, you might need to assign the flow locale to business objects. Table 23 shows the method that the collaboration can use to retrieve the locale associated with the flow.

Table 23. Method to retrieve the collaboration’s flow locale

Collaboration library class	Method
BaseCollaboration	getLocale()

Your collaboration template must ensure that the locales of the business objects are well maintained and properly used during the flow of any collaboration scenarios. Your collaboration can access this locale with the methods shown in Table 24.

Table 24. Methods to access the business object locale

Collaboration library class	Method
BusObj	getLocale(), setLocale()

When Process Designer creates a new port for a collaboration template, it creates a new BusObj object for the port with the name *portName* BusObj, where *portName* is the name of the port. For example, if you create a port named To, Process Designer creates a BusObj object named ToBusObj with code that looks as follows:

```
BusObj ToBusObj = new BusObj("Item");
```

The constructor of the BusObj class creates a BusObj object with its locale set to the collaboration locale. If the business object needs to associate its data with the flow locale, the collaboration template must modify the business object's locale.

For example, suppose Process Designer generates code in Figure 23 to create BusObj objects for two ports, To and From.

```
BusObj ToBusObj = new BusObj(triggeringBusObj.getType());
BusObj FromBusObj = new BusObj(triggeringBusObj.getType());
```

Figure 23. Generated code to create business objects for ports

The following code fragment internationalizes the generated code in Figure 23, by ensuring that the flow locale is set in these new BusObj objects:

```
BusObj ToBusObj = new BusObj(triggeringBusObj.getType());
BusObj FromBusObj = new BusObj(triggeringBusObj.getType());

// get flow locale from BaseCollaboration
triggerLocale = getLocale();

// set newly created BusObj objects' locale to flow locale
ToBusObj.setLocale(triggerLocale);
FromBusObj.setLocale(triggerLocale);
```

The BusObj() constructor also accepts a locale name as an argument. Therefore, an alternative way to rewrite the generated code in Figure 23 is to pass the flow locale directly to the constructor call, as follows:

```
// get flow locale from BaseCollaboration
triggerLocale = getLocale();

BusObj ToBusObj = new BusObj(triggeringBusObj.getType(), triggerLocale);
BusObj FromBusObj = new BusObj(triggeringBusObj.getType(), triggerLocale);
```

Note: The copy() and duplicate() methods of the BusObj class automatically handle assignment of the business object locale. Therefore, if the source business object has the correct locale, the target business object will have this locale as well.

Collaboration configuration properties: As discussed in “Defining collaboration configuration properties (the Properties tab)” on page 91, a collaboration template can use two types of configuration properties to customize its execution:

- Standard configuration properties are available to all collaborations.
- Collaboration-specific configuration properties are unique to the particular collaboration template in which they are defined.

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

The collaboration template obtains the values of configuration properties with the `getConfigProperty()` or `getConfigPropertyArray()` method of the `BaseCollaboration` class. These methods correctly handle characters from multibyte code sets. However, to ensure that your collaboration template is internationalized, its code must correctly handle these configuration-property values once it retrieves them. The collaboration template must *not* assume that configuration-property values contain only single-byte characters.

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

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

Character-encoding design principles

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

- Universal multiple octet Coded Character Set: UCS-2
The UCS-2 encoding is the Unicode character set encoded in 2 bytes (octets).
- UCS Transformation Format, 8-bit form: UTF-8
The UTF-8 encoding is designed to address the use of Unicode character data in UNIX[®] environments. It supports all ASCII code values (0..127) so that they are never interpreted as anything except a true ASCII code. Each code value is usually represented as a 1-, 2-, or 3- byte value.

Most components in the WebSphere business integration system, including InterChange Server and its collaboration runtime environment, are written in Java. Therefore, when data is transferred between a collaboration and other components within InterChange Server, it is encoded in the Unicode code set and there is no need for character conversion.

Designing collaborations for bidirectional scripts

WebSphere Business Integration Server supports bidirectional scripts. This support is in a standard Windows-type bidirectional format (logical left to right). Because of this support, all collaborations also support bidirectional scripts. However, data entering a collaboration may come from:

- An adapter that supports bidirectional scripts. To determine if an adapter supports bidirectional scripts, see your adapter user guide.

- A component that does not support bidirectional scripts, such as the Server Access Interface, an adapter that does not support bidirectional scripts, or data imported from an external source where the bidirectional support is unknown.

Bidirectional format inconsistencies cause comparisons within a collaboration to return incorrect results. These types of errors can be avoided by following these guidelines:

- Accept input only from sources that enforce the same bidirectional format as WebSphere Business Integration Server such as the adapters that are already enabled with this support.
- Enable the connectors that use this collaboration to enforce the correct bidirectional format (see “Enabling connectors for bidirectional scripts”).
- Use the APIs in the CxBidiEngine class to transform all data into a consistent bidirectional format (see Chapter 28, “CxBidiEngine class,” on page 419).

Enabling connectors for bidirectional scripts

To enable a connector to support a bidirectional script:

1. Set the BiDiTransformation property to true on the **Standard** tab of Connector Configurator. (See Figure 24). Setting the value to true allows the connector designer to display the other parameters that support bidirectional scripts for the connector.

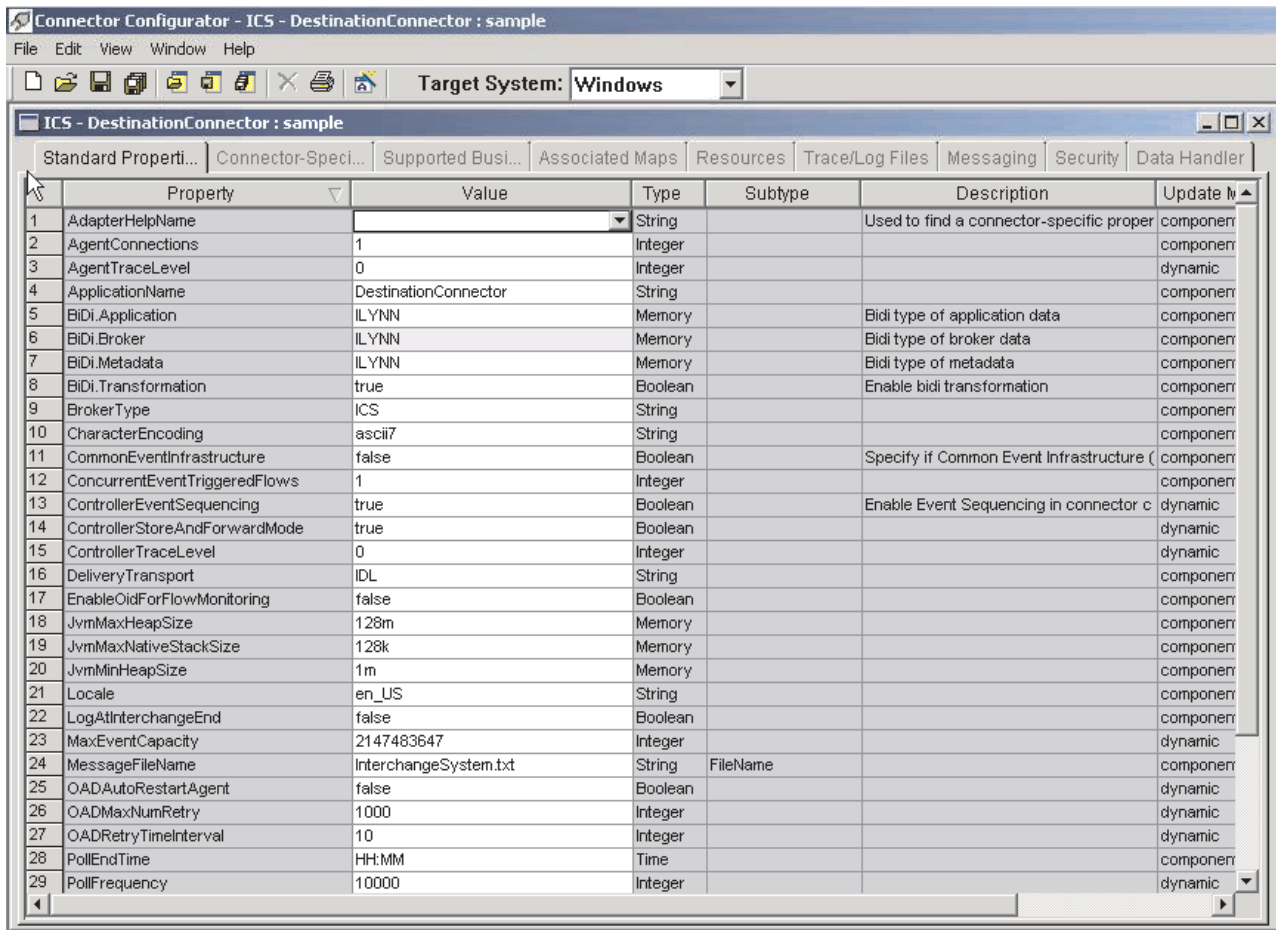


Figure 24. The BiDiTransformation property in the Connector Configurator

2. Specify the bidirectional options for the following:

- BiDi.Application
- BiDi.Broker
- BiDi.MetaData

Each property displays a dialog (see Figure 25) in which you choose the bidirectional parameters to support. See Table 25 for a description of the parameters.

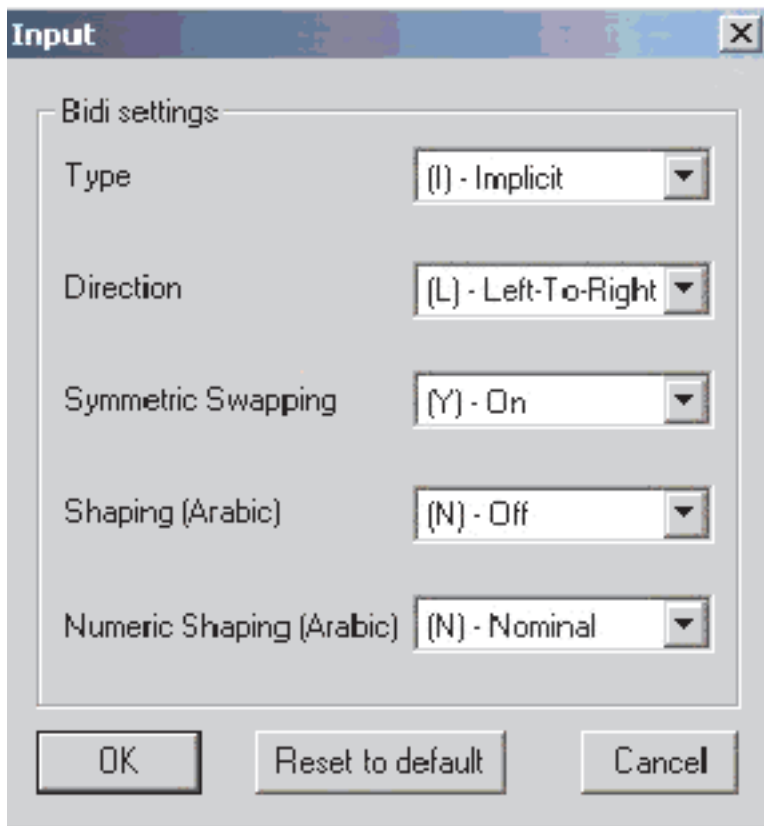


Figure 25. bidirectional script parameter input dialog

Table 25. Values for bidirectional format strings

Letter position	Purpose	Values	Description	Default
1	Type	I	Implicit (Logical)	I
		V	Visual	
2	Direction	L	Left to Right	L
		R	Right to Left	
3	Symmetric Swapping	Y	Symmetric swapping is on	Y
		N	Symmetric swapping is off	
4	Shaping	Y	Text is shaped	N
		N	Text is not shaped	
5	Numeric Shaping	H	Hindi	N
		C	Contextual	
		N	Nominal	

3. Deploy the connector. (See the *WebSphere InterChange Server Implementation Guide* for more information about deploying connectors.)

Configuring database connection resilience feature

Many times a collaboration is critical to the success of an enterprise. In these cases, the availability of databases involved in the collaboration is crucial. However, there may be occasions when the databases may be unavailable for periods of time (for example, database failover or routine maintenance). The database connection

resilience feature provides protection against these temporary periods of when a database may be unavailable. The following sections describe what is involved to insure critical databases are available to your collaborations. Note that database connection resilience does not directly affect collaboration design.

Customizing the database connection resilience feature

To customize the database connection resilience feature, in the InterChange Server.cfg file, you can set the DB_CONNECT_RETRIES and DB_CONNECT_INTERVAL properties for each database used by InterChange Server for event management, transactions, repository, flow monitoring, relationships, and user registry:

- DB_CONNECT_RETRIES - specifies the maximum number of times InterChange Server retries the database connection.
- DB_CONNECT_INTERVAL - specifies the amount of time InterChange Server waits between retries.

The global default properties for all of the databases that use these properties are defined in the DB_CONNECTIVITY section of the file.

You can set defaults for DB_CONNECT_RETRIES and DB_CONNECT_INTERVAL for individual databases in the sections of the InterchangeServer.cfg file shown in Table 26. Configuring defaults for the databases separately provides flexibility when the system attempts to reconnect to the databases.

Table 26. Sections of InterchangeSystem.cfg to set database resilience defaults

Section	Database
EVENT_MANAGEMENT	Event management
FLOW_MONITORING	Flow monitoring
REPOSITORY	Repository
TRANSACTIONS	Transactions
USER_REGISTRY	User registry

Note: You can configure the defaults for the relationships database only in the DB_CONNECTIVITY section of the InterchangeSystem.cfg file.

Set the DB_CONNECT_RETRIES and DB_CONNECT_INTERVAL properties in the Server Configuration in System Manager. These properties are set to their default values even if they are not defined in the InterchangeSystem.cfg file. To customize the properties:

1. Connect to InterChange Server using System Manager.
2. Open the Server Configuration in the System Manager.
3. Set the DB_CONNECT_RETRIES and DB_CONNECT_INTERVAL properties.
4. Save the changes.
5. Restart InterChange Server.

Set the DB_CONNECT_INTERVAL property to a value that is neither too small nor too large. If the value is too large, for example 10 minutes, after one unsuccessful attempt to recover database connections, the next attempt begins after ten minutes even though the database may become available one minute later. If the value is too small, for example 2 seconds, in that same 1-minute period InterChange Server attempts to recover the database connection 30 times, and all attempts fail.

If scheduled database outages cause a cold restart of the database, set `DB_CONNECT_RETRIES` and `DB_CONNECT_INTERVAL` to values that can accommodate the outage, if it is known. Setting the properties to account for the downtime prevents unnecessary recovery attempts.

Two other properties relate to database connection resilience for all user databases: `maxConnectRetries` and `maxConnectRetryInterval`. These properties are defined in each user-defined database connection. These properties work the same as `DB_CONNECT_RETRIES` and `DB_CONNECT_INTERVAL` and cause InterChange Server to attempt to reconnect to user databases as it does for the system databases. To configure these properties:

1. Open System Manager.
2. Open or create a database connection.
3. Set the `maxConnectRetries` and `maxConnectRetryInterval` properties.
4. Save the user database connection.
5. Deploy the user database connection to InterChange Server.

Default settings for database connection resilience

After InterChange Server installation, the default value of `DB_CONNECT_RETRIES` is 3 and the value for `DB_CONNECT_INTERVAL` is 60 seconds. You can change these defaults by editing the property values in the `DB_CONNECTIVITY` section of the `InterchangeSystem.cfg` file. You can also set the values in the Database Connectivity dialog of System Manager (see Figure 26).

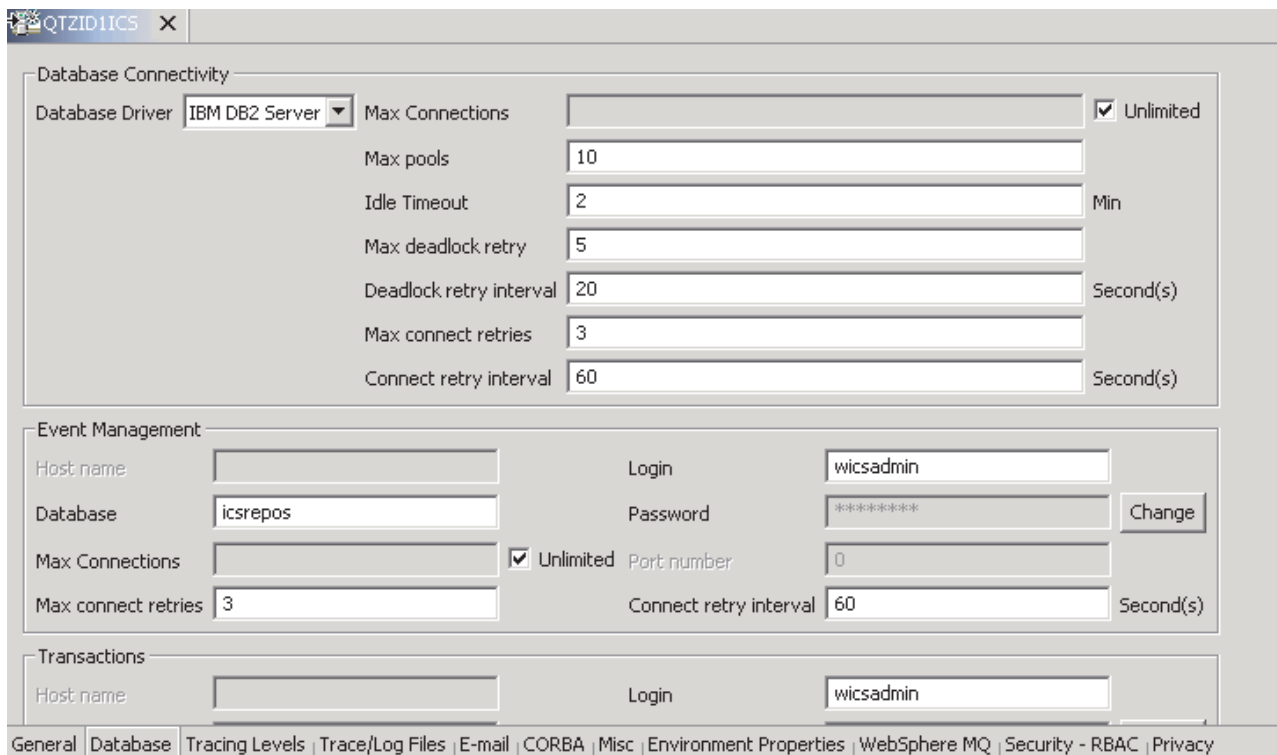


Figure 26. System Manager showing default values for database resilience

Handling failed events

In some cases, you may need to handle failed events within your collaboration instead of using the Flow Manager. For example, you may want to save off only those failed flows in a batch process for further management. There are APIs that allow a running collaboration to automatically handle failed events during collaboration processing. These APIs allow you to query, save, drop or resubmit failed events inside the collaboration. The APIs provide more flexibility than using the Flow Manager because you can build the processing within the business flow to automatically handle the failed events. This functionality allows a batch process to continue while providing the ability to resubmit or delete the failing events asynchronously.

Use the listed methods to provide your collaboration with failed event handling:

- “queryFailedEvents()” on page 355 – retrieves failed event records from the failed event database. Each event can have one of the following states:
 - PENDING
 - USER_GENERATED
 - DEFERRED_RECOVERY
 - SVC_CALL_IN_TRANSIT
 - DELIVERY_POSSIBLE_DUPLICATE
 - SVC_CALL_WAITING (for long-lived business processes)

Note: The Flow Manager displays failed events of type USER_GENERATED as USER_SAVED.

- “saveFailedEvent()” on page 362 – saves failed events for later processing.
- “dropFailedEvent()” on page 343 – deletes failed events from the database.
- “resubmitFailedEvent()” on page 360 – resubmits failed events for reprocessing.

Saving a failed event

While handling errors during collaboration execution, you may decide that certain conditions require outside intervention before processing the event. In these cases, your collaboration logic could save the event for analysis and resubmission later. This capability allows the collaboration to complete processing objects that are correct to prevent delays in the overall process. For example, an order delivery system continues processing the correct items in an order so the warehouse can pull the good items without the incorrect items holding up the entire order.

Figure 27 on page 76 shows the logic for an order delivery collaboration designed to process orders even if there are items in error. Notice that instead of saving the items immediately, the collaboration adds a failed business object to the parent business object and then returns success. When control returns to the main diagram, any existing children of the parent business object are saved using the saveFailedEvent() method on page 362.

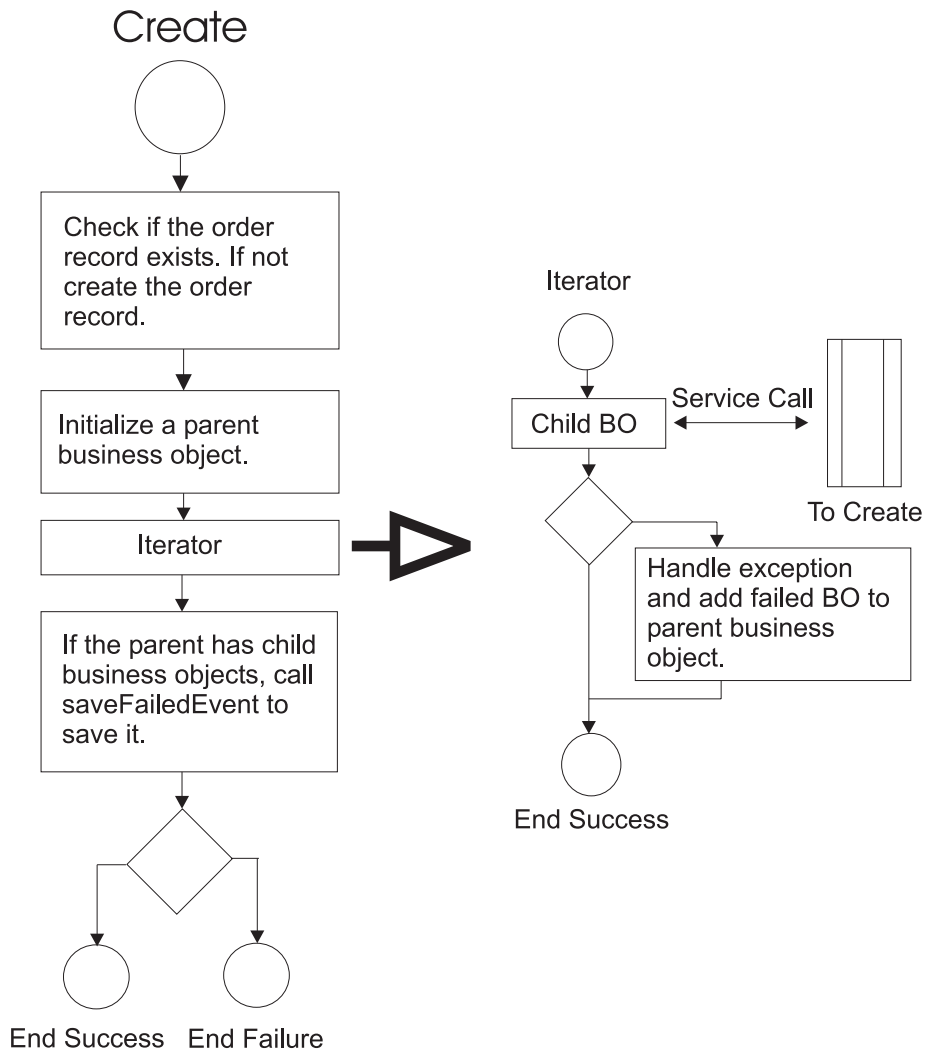


Figure 27. Order delivery collaboration with the ability to save erroneous items

You could also build a parallel process to look for failed events, correct those failures and then resubmit them for processing. For additional information see:

- “queryFailedEvents()” on page 355
- “dropFailedEvent()” on page 343
- “resubmitFailedEvent()” on page 360

Coding to save a failed event

The following example implements the logic shown in Figure 27.

Declaration:

```
// Define an empty parent B0:
BusObj iterOrderItem = new BusObj("OrderItem");
BusObj parentBo = new BusObj("SalesOrder");
```

In the action step to initialize the parent business object:

```
// Set values for parent B0:
// Set the values which are the same as triggeringBusObj
parentBo.set("SalesOrderId",triggeringBusObj.getString("SalesOrderId"));
parentBo.set("CustomerId",triggeringBusObj.getString("CustomerId"));
```

```
parentBo.set("OrderDate",triggeringBusObj.getString("OrderDate"));
//Set Verb to the parent bo
parentBo.setVerb(triggeringBusObj.getVerb());
```

In the iteration action step to save the child business object:

```
//Add the failed business object to parent bo
parentBo.set("OrderItems",iterOrderItem);
```

In the action step to check and save the parent business object:

```
// Check to see if the parent bo contains at least one child bo
BusObjArray boArray = parentBo.getBusObjArray("OrderItems");
int nChildBo = 0;
if (boArray != null)
    nChildBo = boArray.size();

if(nChildBo>0) {
    // Call collaboration API to save the failed event
    saveFailedEvent(parentBo);
}
```

Enabling a collaboration to save failed events

To enable a collaboration to save failed events, update the collaboration template with an activity definition to detect and save the failed event. You do this directly in the action block or in the Activity Editor in the Java view.

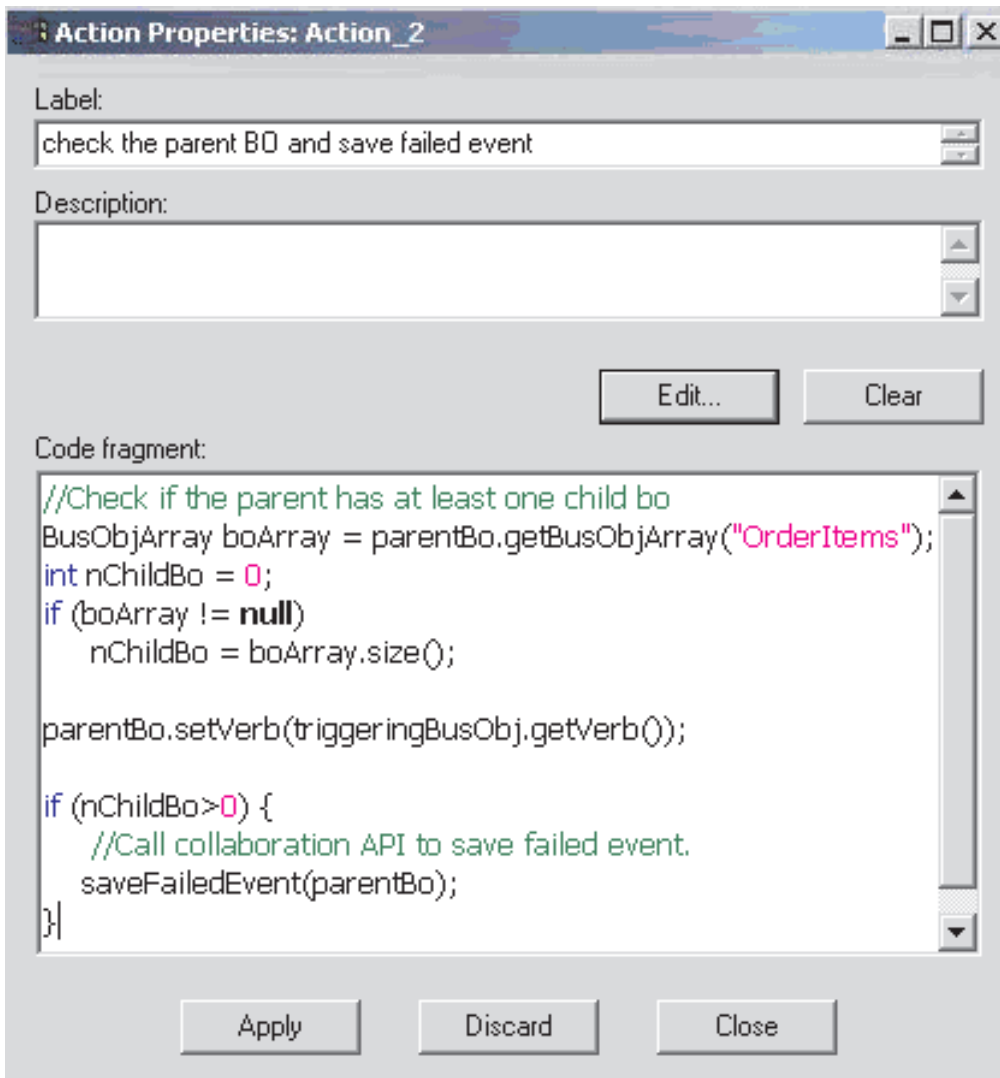


Figure 28. Saving a failed event

Note: There are no function blocks in the Activity Editor for `queryFailedEvents()`, `dropFailedEvent()`, `resubmitFailedEvent()` or `saveFailedEvent()`.

Required data structures

To successfully implement failed event handling in your collaborations, you must import the following data structures:

- `CxCommon.EventManagement.FailedEventInfo;`
- `CxCommon.EventManagementEventQueryDef;`
- `CxCommonEventManagement.EventKeyAttrDef;`

These structures are automatically imported into your collaboration when compiling collaboration templates. For more information about the structures, see Chapter 30, “EventManagement package,” on page 429.

Resubmitting failed events

After saving failed events, your collaboration may resubmit those events for reprocessing. The InterChange Server uses the `USER_GENERATED` status to differentiate these failed events from other failed events that are not saved by the

collaboration. When a collaboration resubmits a `USER_GENERATED` event the ICS changes the status of the event. The collaboration then processes the event anew. If the event fails again, the status is changed to `USER_GENERATED` and not `FAILED` so the collaboration can determine how to proceed in recovering from the second failure.

Note: You can also use Flow Manager to manually locate and resubmit these events.

Chapter 4. Building a collaboration template

This chapter describes how to create and modify a collaboration template definition. You must perform the following tasks:

1. Create the template definition. See “Creating a collaboration template” for more information.
2. Provide information about the template’s properties. See “Providing template property information” on page 83 for more information.
3. Define a scenario. See “Defining scenarios” on page 96 for more information.
4. Create an activity diagram for the scenario you defined. See “Creating an activity diagram” on page 100 for general information, and then refer to Chapter 5, “Using activity diagrams,” on page 109 for detailed instructions.
5. Define additional scenarios as needed, and create the associated activity diagrams.
6. Create a message file for the template. See “Creating the message file” on page 100 for more information.
7. Compile the collaboration template. See “Compiling a collaboration template” on page 101 for more information.
8. Optionally, test the collaboration. You can use the Test Connector in the Integrated Testing Environment to verify that your collaboration works as planned. See “Testing a collaboration” on page 107 for more information.

If you have a template that was created with a previous version of Process Designer, you can convert that template into the format required by version 4.2. For conversion instructions, see “Converting templates” on page 102.

Creating a collaboration template

Use Process Designer to create, edit, and compile a collaboration template. The following sections describe how to define a new template and provide the basic information it requires.

You must provide certain information prior to building the template; other types of information can be supplied at any time during development. The following information is required for template creation:

Template name	See “Creating the template definition” on page 82
Ports	See “Defining ports and triggering events (the Ports and Triggering Events tab)” on page 94
Scenario definitions	See “Defining scenarios” on page 96

The following information is optional and you can supply it at any time during the development process:

Support for long-lived business processes	See “Adding support for long-lived business processes” on page 84
Package name	See “Specifying a collaboration package” on page 86

Minimum transaction level	See “Specifying the minimum transaction level” on page 84
Configuration properties	See “Defining collaboration configuration properties (the Properties tab)” on page 91
Template variables	See “Declaring and editing template variables (the Declarations tab)” on page 86
Import statements	See “Importing Java packages” on page 87

Creating the template definition

To create a new collaboration template, do the following from within System Manager:

1. Right-click the Collaboration Templates folder for your project, and then click the Create New Collaboration Template option. Process Designer opens and displays the New Template dialog box, as shown in Figure 29.

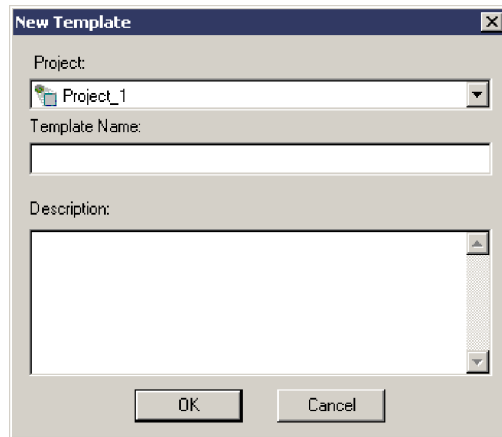


Figure 29. New Template dialog box

2. In the Project field, use the drop-down list to select the name of the Integration Component Library user project the template belongs to.
3. Type the name of the template in the Template Name field. A template name can include alphabetic characters, numbers, and underscores. Because Process Designer creates a source file (.java) and a Java class file (.class) based on the template name, it is recommended that you follow these Java class naming conventions:
 - Start a collaboration template name with an uppercase letter.
 - If the name contains multiple words, start each word with an uppercase letter (for example, CustomerSync).
 - Do not embed spaces in the name.
4. If desired, include a brief description of the template in the Description field.

Note: Do not include a hard return (a carriage return) in the template description.

5. Click OK. Process Designer opens and displays the new template and its top-level tree in its Template Tree pane.

Providing template property information

A collaboration's Template Definitions window provides the four tabs listed in Table 27 for defining a collaboration template's properties.

Table 27. Tabs of the Definitions window

Template Definitions tabs	Description	For more information
General	Enables you to define the following information for a collaboration template: <ul style="list-style-type: none">• Template description• Support for Long-lived business processes (LLBP)• Minimum transaction level• Package information	"Defining general property information (the General tab)" on page 83
Declarations	Enables you to define template variables and view system-generated template variables.	"Declaring and editing template variables (the Declarations tab)" on page 86
Properties	Enables you to specify the name, type, and value of user-defined collaboration template properties.	"Defining collaboration configuration properties (the Properties tab)" on page 91
Ports and Triggering Events	Enables you to define the ports and triggering events for the collaboration template.	"Defining ports and triggering events (the Ports and Triggering Events tab)" on page 94

Defining general property information (the General tab)

The General tab of the Template Definitions window (see Figure 30) displays general property information for the collaboration template, including that listed in Table 28.

Table 28. General template definition information

General template property	Description	For more information
Description of the collaboration template	This field of a collaboration template is optional. In it, you can enter text that is available to all users of the collaboration template.	None
Support for long-lived business processes	Specifies whether the template supports long-lived business processes.	"Adding support for long-lived business processes" on page 84
Transaction level (for transactional collaborations only)	Sets a minimum transaction level for all operations in the collaboration.	"Specifying the minimum transaction level" on page 84
Collaboration package	The Java package in which the collaboration resides	"Specifying a collaboration package" on page 86

Figure 30 shows the General tab within the Definitions window.

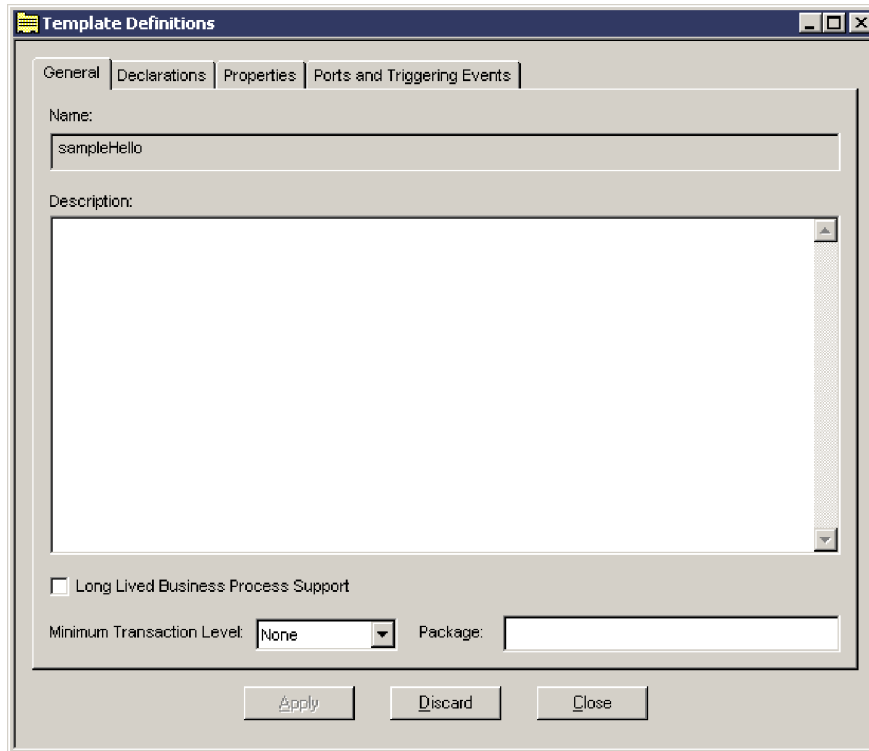


Figure 30. General collaboration properties

Adding support for long-lived business processes

Long-lived business process support enables you to deploy a collaboration as a long-lived business process, and to specify a timeout value for service calls in this environment. In order to use this functionality, you must do the following:

- Select the Long Lived Business Process Support option on the General tab
- Optionally, create a user-defined collaboration template property to represent the timeout value for the service call used in long-lived business processing. See “Defining collaboration configuration properties (the Properties tab)” on page 91 for more information.

Specifying the minimum transaction level

When a collaboration is transactional, InterChange Server rolls back the collaboration upon failure of its transaction. The rollback executes template-defined compensations to reverse the collaboration’s data modifications. For an explanation of transactional collaborations, see “Using transactional features” on page 143.

The transaction level determines the mechanisms by which the collaboration’s scenarios are executed. Collaboration objects execute at one of the transaction levels described in Table 29.

Table 29. Transaction levels

Transaction level	Effect	System behavior
None	Collaboration is not transactional.	If an error occurs during execution of the collaboration, the system just sends it to the log and then terminates execution.

Table 29. Transaction levels (continued)

Transaction level	Effect	System behavior
Minimal Effort	Collaboration is transactional; it has compensations defined for its scenario's subtransactions.	If an error occurs during execution of that scenario, InterChange Server rolls back the scenario, executing the compensation for each subtransactional step.
Best Effort	Does what Minimal Effort does; in addition to compensations, data isolation is used to ensure correctness.	InterChange Server checks that data is virtually isolated for the duration of its use in the transactional collaboration by checking that the data's value has not changed since its previous use. Best-Effort isolation checking leaves a small window of time during the isolation check when the data is vulnerable to changes by other application transactions.
Stringent	Does everything that Best Effort does, but eliminates the data isolation window of vulnerability.	Application locks the data when the isolation is checked. Supported by applications whose APIs support an atomic "test and set" operation.

A collaboration template developer sets the minimum transaction level for collaboration objects created from the template. For example, if a collaboration deals with critical data and you want to ensure it is always rolled back when it fails, you can set its minimum transaction level to Minimal Effort. If you design a collaboration for transactional execution, but it can be used successfully without the transactional features, you can set the minimum transaction level to None.

An administrator can raise the transaction level for a collaboration object if its connectors support the higher transaction level. However, the transaction level for a collaboration object cannot be lower than the minimum specified in the template.

Tip

You can create compensation to permit transactional operation while setting the minimum transaction level of the collaboration templates to None. If greater rigor is needed and the connectors in use can support a higher transaction level, an administrator can raise the transaction level of the collaboration object at bind time. For more information on compensation, see "Defining compensation" on page 131.

To assign a minimum transaction level to the collaboration template, do the following:

1. Ensure the Template Definitions window is open and the General tab is displayed.
2. Use the Minimum Transaction Level pull-down menu to select the minimum transaction level you want to use. If you are editing a collaboration template that is not transactional, keep the default value of None.
3. Click Apply to save the changes.

Specifying a collaboration package

A package is a group of collaborations that have related functions. All collaborations that Process Designer accesses belong to the package `UserCollaborations` or to a subpackage of `UserCollaborations`. Therefore, the `UserCollaborations` package includes:

- Collaborations that are provided with the WebSphere InterChange Server software
- Collaborations that other collaboration developers have created

You can create subpackages under `UserCollaborations` to group custom collaboration templates. For example, if you create several collaboration templates that deal with office supply management, you can create a subpackage called `OfficeSupplyMgmt`. In it, you can put the `PaperClipMgmt` collaboration and the `PencilInventory` collaboration.

If you specify that a collaboration template is part of a package, Process Designer uses the package name to create a subdirectory in your Integration Component Library project's `Template\Classes` directory. (During deployment, the `ProductDir\collaborations\classes\UserCollaborations` directory is created to store the package information.)

The product installation sets the `CLASSPATH` environment variable to include all collaborations under `UserCollaborations` in the class path. Process Designer places a collaboration template's `.class` and `.java` files in the subdirectory.

To specify a package in which to store the collaboration template:

1. Ensure the Template Definitions window is open, and the General tab is displayed.
2. In the Package field, enter the name of the package in which to store the collaboration template.

When you specify the name of an existing package, Process Designer adds the collaboration template to the package. When you specify the name of a package that does not yet exist, Process Designer creates it.

3. Click Apply to save the changes.

You can revise an existing collaboration template definition to add or change a package name at any time.

Declaring and editing template variables (the Declarations tab)

The Declarations tab of the Template Definitions window displays information about the template variables of the collaboration template. *Template variables* are collaboration variables whose scope is all scenarios in a collaboration; that is, a template variable is global to all scenarios in a collaboration. (They are comparable to class variables in the Java programming language.) For example, a collaboration that involves customer transactions can have a `customerID` template variable that identifies the customer across all scenarios. You can create or modify template variables at any time during development.

Figure 31 shows the Declarations tab within the Template Definitions window.

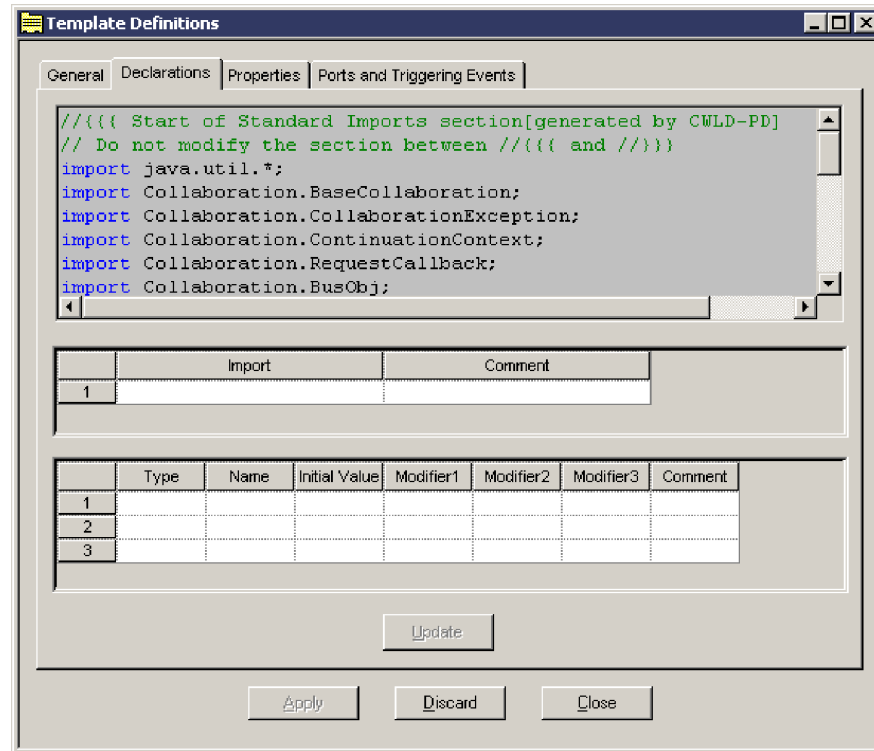


Figure 31. Declarations tab of the Template Definitions window

Within the Declarations tab, you can do any of the following tasks:

- Specify the code for any import statements. See “Importing Java packages” for more information.
- Type the declaration text for user-defined template variables. See “Declaring template variables” on page 90 for more information.
- View system-generated template variables (which you cannot edit). See “System-generated variables” on page 91 for more information.

Importing Java packages

You can use the Declarations tab to import specific Java classes into the collaboration. A Java class imports a package of other classes to gain access to their functions. For example, a class imports the packages `java.math`, `java.security`, and `java.text` to use their arithmetic, security, and internationalization functions, respectively. Because a collaboration template is a class, it can use classes or groups of classes (called packages) supplied by the Java Development Kit or from third-party products.

All Java classes, by default, implicitly import the classes in the package `java.lang`. In addition, Process Designer implicitly imports the classes in the package `java.util` for use in all collaboration templates.

The following import statement imports the `java.math` classes from the JDK. (The asterisk indicates to import all classes within the specified package.):

```
java.math.*;
```

Alternatively, the following statement imports just the package’s `BigDecimal` class:

```
java.math.BigDecimal;
```

You can add import statements to your code at any time during the development of a collaboration.

To import Java classes:

1. Ensure the Template Definitions window is open and that the Declarations tab is displayed.
2. Place the cursor in left heading cell of the import table. Right-click and select Add, as shown in Figure 32. A new row is added to the table.

Note: You can also add a new row by clicking on the last row currently in the table.

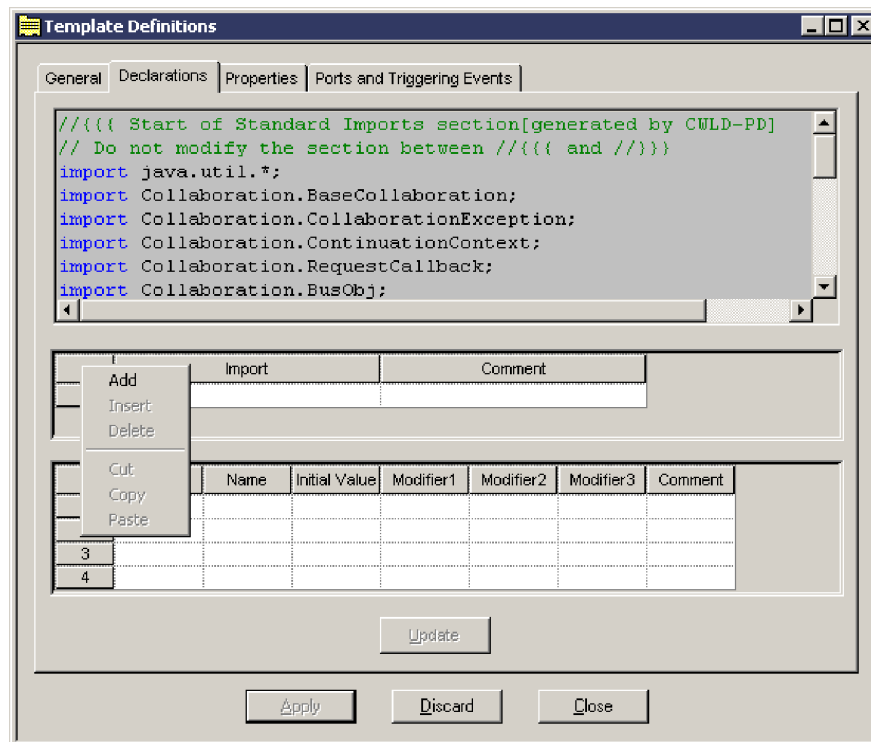


Figure 32. Adding an import statement

3. Type the import statement in the Import column. For example:
`java.math.*`
4. Optionally, enter a brief description of the import statement in the Comment column.
5. Click Apply to save the changes.
6. Repeat step 2 through step 5 to add additional import statements. You can have an unlimited number of import statements in a collaboration template.

If the imported classes are in a third-party package rather than in the JDK, you must edit the `ProductDir\bin\cwtools.cfg` file [codeGeneration] section to reflect the package path before you compile your template.

Before you deploy a collaboration that uses classes imported from a third-party package, you must update the JCLASSES variable in the system on which the collaboration is deployed. If the imported classes are in a third-party package rather than in the JDK, you must add them to the path of the imported classes in

the JCLASSES variable. IBM recommends that you use some mechanism to differentiate those classes in JCLASSES that are standard from those that are custom. For example, you can create a new variable to hold only those custom classes and append this new variable to JCLASSES, as follows:

1. Place the CwMacroUtils.jar file in its own directory. For example, create a \dependencies directory below the product directory and place the .jar file in it.
2. Edit the file used to start ICS (by default, *ProductDir*\bin\start_server.bat or *ProductDir*/bin/CWSharedEnv.sh) to include the new path for the CwMacroUtils.jar file. Add the following entry to the file:

```
set DEPENDENCIES=ProductDir/dependencies/CwMacroUtils.jar
```

where *ProductDir* is the location in which WebSphere InterChange Server is installed.

3. Add the new DEPENDENCIES variable to the JCLASSES entry as follows, depending on your operating system.
On a UNIX system, use the following syntax, where *ExistingJarFiles* represents the .jar files already included in JCLASSES:

```
set JCLASSES = $JCLASSES:ExistingJarFiles:$DEPENDENCIES
```


On a Windows system, use the following syntax, where *ExistingJarFiles* represents the .jar files already included in JCLASSES:

```
set JCLASSES = ExistingJarFiles;%DEPENDENCIES%
```
4. In each collaboration that uses the classes, include the *PackageName.ClassName* specified in the CwMacroUtils.jar file.
5. Restart ICS to make the methods available to the collaborations.

When importing a custom class, you can get an error message indicating that the InterChange Server software was unable to find the custom class. If this occurs, check the following:

- Ensure that the custom class is part of a package. It is good programming practice for custom classes to be placed in a package. Make sure that the custom class code includes a correct package statement and that it is placed at the beginning of the source file, prior to any class or interface declarations.
- Verify that the import statement is correct in the collaboration template. The import statement must reference the correct package name; it can further specify the name of the custom class or it can reference all classes in the package. For example, if your package name is COM.acme.graphics and the custom class is Rectangle, you can import the entire package:

```
COM.acme.graphics.*;
```


Or, you can import just the Rectangle custom class:

```
COM.acme.graphics.Rectangle;
```
- Be sure that you have updated the CLASSPATH environment variable to include the path to the package containing the custom class, or to the custom class itself if it is not in a package.

For example, when importing a custom class, you can create a folder called *ProductDir*\lib\com\crossworlds\package, where *ProductDir* is the location in which WebSphere InterChange Server is installed and *package* is the name of your package. Then, place your custom class file under the folder you just created. Finally, in the CLASSPATH variable in the start_server.bat file, include the path *ProductDir*\lib.

Declaring template variables

You can also use the Declarations tab to declare your own template variables that are used by the collaboration.

To use a variable, you must first declare it by specifying its type and name. A variable in a collaboration template can be one of the following data types:

- A basic, or primitive, data type, such as a byte, short, int, long, float, double, char, or boolean
- A Java class, such as String or Integer
- An InterChange Server-defined class, such as BusObj, BusObjArray, or CollaborationException
- A class that you define, if you are an advanced user

Note: LongText and Date are product-specific designations for special-purpose strings in business object attributes. Use the String data type in your code to represent a variable for a business object attribute whose type is LongText or Date.

To declare template variables, do the following:

1. Ensure the Template Definitions window is open and the Declarations tab is displayed.
2. Place the cursor in left heading cell of the variable table. Right-click and select Add. A new row is added to the table.

Note: You can also add a new row by clicking on the last row currently in the table.

3. Use the drop-down menu in the Type column to specify the type of variable you want to declare.
4. Specify the variable's name in the Name column.
5. Specify the variable's initial value in the Initial Value column.

Note: String values must be entered in quotation marks (for example, the string Yes must be entered as "Yes.")

6. Specify any modifiers you want to apply to the variable (for example, public, private, protected) in the Modifier1, Modifier2, and Modifier3 columns. Note that you do not have to specify a modifier in all three columns.

Note: Do not use the modifier Static when defining template variables.

7. Click Update to add the new variable to the list of declarations at the top of the tab, and then click Apply to save the changes.

You can declare variables whose values remain persistent across multiple invocations of a collaboration. Suppose you want to keep a counter of an action within the collaboration, and you want this counter to be incremented with each separate run of the collaboration. Use the variable table in the Declarations tab to create a integer variable named ctr that is public.

Later, within the collaboration code itself, increment the counter:

```
ctr = ctr+1;
```

The ctr variable increases with each collaboration execution.

Special considerations for template variables used with long-lived business processes: If a collaboration is to be deployed as a long-lived business process, ensure that all variables you want to persist are defined as global template variables or global port variables.

In addition, ensure that those variables are of one of the following types:

- Java serializable data types, including byte, short, int, long, float, double, char, boolean, string, Integer, or any user-defined data type that implements the Java Serializable or Externalizable interface
- WebSphere InterChange Server BusObj data type
- WebSphere InterChange Server BusObjArray data type

Variables of other types do not persist in a long-lived business process.

System-generated variables

Process Designer automatically declares the following collaboration variables:

- Two collaboration variables that are available in all collaborations: triggeringBusObj and currentException.
- One variable for each port.

Table 30 lists and describes these system-generated variables.

Table 30. System-generated variables

Variable	Description
triggeringBusObj	The triggeringBusObj variable contains the flow trigger (triggering event or triggering access call) for a scenario. The flow trigger is a business object and a verb. A triggering event represents an application event and its data. The arrival of the flow trigger starts the execution of a scenario. This variable is a template variable; that is, its scope is the entire collaboration.
currentException	The currentException variable contains an exception object raised by the immediately preceding action, subactivity, or iterator. Process Designer implicitly declares currentException, whose scope is the action that immediately follows the raising of an exception. A scenario must check the value of currentException on the transition link or code fragment that immediately follows the activity that generated the exception.
Port Variables	Process Designer declares a template variable for the business object associated with each port in the collaboration. These generated declarations are visible under the Declarations tab of the Template Definitions window. The name of each port variable is the name of the port with BusObj appended. For example, if the port name is SourceInvoice, the variable name is SourceInvoiceBusObj. The declaration also instantiates a BusObj of the same type for which the port is defined. It initially sets the attributes of the business object to null. You can use these port variables to handle the triggering event. For more information on this, refer to “Copying the triggering event” on page 205.

Defining collaboration configuration properties (the Properties tab)

Collaboration templates have two types of configuration properties:

- *Standard properties* provide information that all collaborations need, such as tracing level and an email address for message notifications. All collaborations

have the same standard configuration properties, which are defined by WebSphere InterChange Server. See “Standard properties for collaboration templates” on page 468 for more information.

- *Collaboration-specific properties* are optional; they are defined by a collaboration developer. The collaboration uses the value of the property to determine an aspect of its behavior. Properties can be any of the following types:
 - Date
 - Double
 - Float
 - Integer
 - Boolean
 - String
 - Time
 - URL

An administrator works with both types of properties when configuring a collaboration.

As a collaboration developer, you decide whether a collaboration needs collaboration-specific configuration properties. If it does, you define their names and default values. These configuration properties enable a collaboration user to specify data that influences how the collaboration behaves.

Table 31 provides some examples of the types of properties you can create.

Table 31. Examples of collaboration-specific configuration properties

Type of property	Example
A value that the collaboration uses to set the value of an attribute.	A collaboration can request an application to generate invoices for customers. The collaboration can set the value of a Rate attribute in an Invoice business object. If the collaboration has a property called BILLING_RATE, an administrator can raise or lower the rate based on the current business practice.
Value of true or false, which determines whether the collaboration takes a particular execution path.	WebSphere InterChange Server collaborations that synchronize changes to entities across applications generally have a property called CONVERT_CREATE. When the collaboration receives an Update event, it checks the destination application for the entity to be updated. If the entity does not exist, the collaboration checks the value of the CONVERT_CREATE property. If the property is set to true, the collaboration converts the Update request to a Create request.

The use of collaboration-specific configuration properties is optional, and you can use an unlimited number of them in a template. You can add these properties at any time during development. If you know at the outset the properties that the collaboration needs, you can create them before modeling scenarios. However, when you are in the midst of scenario modeling, you can define additional properties to support the collaboration’s logic.

To create a collaboration-specific configuration property for the collaboration template:

1. Ensure the Template Definitions window is open and that the Properties tab is displayed.

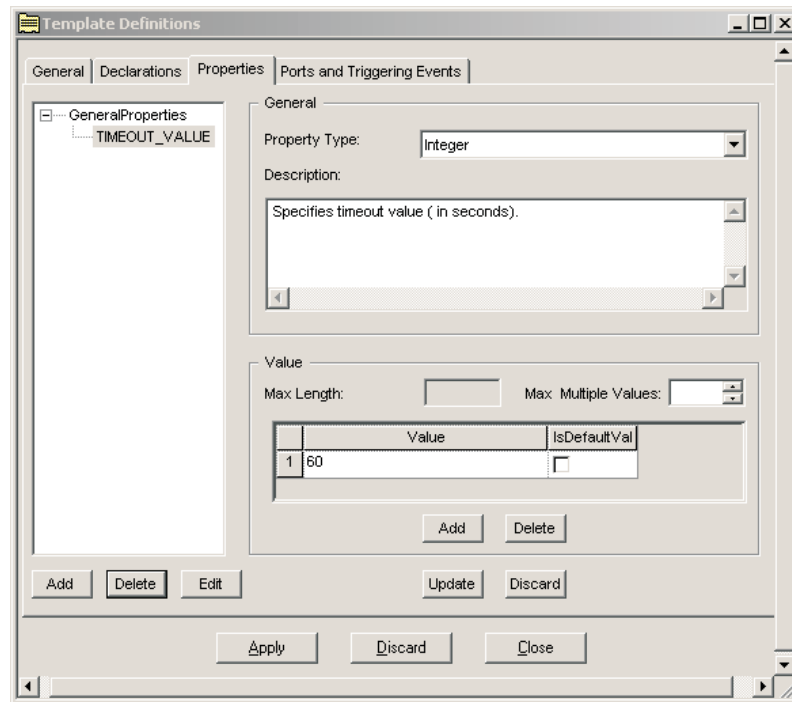


Figure 33. Adding collaboration-specific properties

2. Click the Add button to create a configuration property. The Name dialog box opens.
3. Type the property's name in the Name field, and then click OK.

Note: By convention, configuration property names are uppercase and use underscores to separate words. IBM recommends that configuration property names clearly communicate the purpose or function of the property, because administrators need to read and understand each property.

4. Use the Property Type drop-down menu to select the property's type.
5. If desired, provide a description of the new property in the Description field.
6. If the property type is a string, specify a value in the Max Length field.
7. Optionally, use the Max Multiple Values field to specify the maximum number of multiple values accepted for the property. Note that the number you specify in this field also limits the number of default values the property can have. For example, if you set Max Multiple Values to 2, you can have only two default values for the property, regardless of how many possible values are associated with the property. If you do not specify a value in this field, the default is 1.

Note: The Max Multiple Values attribute of a collaboration-specific property is not often used. Most collaboration-specific properties accept only a single value.

8. Click Add in the Value pane. A new row is added to the table, and the Property Value dialog box opens.

9. Enter a value in the Value field, or specify a range of values in the Range From field, and then click OK. The dialog box closes and the Value column is populated with the information.
10. If the value is a default value, click the checkbox in the IsDefaultValue column.
11. Repeat step 8 on page 93 through step 10 for each value you want to add to the property definition.
12. Click Update.
13. Repeat step 2 through step 12 to add as many configuration properties as you need.
14. When you are finished adding configuration properties, click Apply to save the changes.

To delete a collaboration-specific configuration property, select the name of the property from the list in the left pane of the tab, then click Delete.

Adding properties to support long-lived business processes

If you want to support long-lived business processes (LLBP) with dynamic service call timeout values, you can use either a Java variable or a collaboration-specific property. If you want to use a collaboration-specific property, you must create it when defining the collaboration template. The use of collaboration-specific properties enables the timeout value to be set during runtime, rather than using a static value provided during the initial creation of the service call. Use an integer data type when creating properties for dynamic timeout values.

For example, if you plan to have a service call from the To port that sends a business object with a create request, you can define a collaboration property called CreateTimeout. When you define the service call, use the CreateTimeout property to specify the point at which that service call times out. For details on creating service calls, see “Service calls” on page 126.

Note that you can also use a fixed timeout value that is specified during the creation and definition of a service call; in this situation, no collaboration property is needed. See “Defining the service call type” on page 129.

Defining ports and triggering events (the Ports and Triggering Events tab)

The Ports and Triggering Events tab of the Template Definitions window displays information about the following:

- The ports for the collaboration

In a collaboration template, a *port* is a variable that represents a business object that the collaboration object receives or produces at runtime.
- The triggering event for the collaboration

A business object represents the triggering event or action. When a collaboration receives a business object from a connector, it usually responds with an action. These received business objects are referred to as triggers or triggering events.

Note: The general term for a business object that a collaboration receives is a flow trigger. When a collaboration receives a business object from a connector, this flow trigger is a triggering event. When a collaboration receives a business object from an access client, this business object is referred to as a triggering access call. A port whose business object is

associated with a triggering access call is defined in the same manner as one associated with a triggering event.

For detailed information on using the Ports and Triggering Events tab to define a triggering event, see “Assigning triggering events to scenarios” on page 97.

When a collaboration completes an operation, it usually sends a business object to the connector that initiated the action. Thus, WebSphere InterChange Server often refers to ports in terms of triggering or sending events.

Note: If you add, modify, or delete a business object to or from the repository using Business Object Designer or System Manager, the list of business object definitions displayed in Process Designer is automatically updated. You do *not* have to restart System Manager or Process Designer to see the results of dynamic changes in the business-object field of the Ports and Triggering Events table of the Template Definitions window.

Important

IBM recommends use of this dynamic update feature only in a development environment. Possible complications can result from updating a business object. Dynamic update can impact other functionality in the system, including how to process any events that use the old business object definition and how to resubmit unresolved flows that were originally submitted on the old business object definition. These and other scenarios can cause a mismatch between the business object definitions being processed and the business object definitions in memory. Therefore, in the production system, IBM recommends that you perform updates to business object definitions only when no events are being processed on the system.

For more discussion of collaboration ports, refer to the chapter on collaborations in the *Technical Introduction to IBM WebSphere InterChange Server*.

Creating a port

To create a port, do the following:

1. Ensure the Template Definitions window is open and the Ports and Triggering Events tab is displayed.

At the top of the window is a table that contains port names, business object types, and verbs. The table is empty if you have not yet created a port for this collaboration template.

2. Click Add Port to add a new port to the Ports table.
3. Enter the port name in the Port column of the table.

Follow these guidelines for defining a name for a port:

- Begin the name with an alphabetic character and use only alphanumeric characters and the underscore symbol in the name.
- Although port names are not case-sensitive, you must always refer to the port name using the case in which you defined it.
- In general, it is useful to assign port names that help you remember the port’s purpose. You use the port names throughout the development process, self-explanatory port names make the development effort easier.
- Collaboration developers often create a port name by combining the business object type and its role designation, such as “In” and “Out” or “Source” and

“Destination.” For example, you can call a port SourceCase to indicate that it is the port to the source application and that it is configured for Case business objects.

4. Select the port’s type from the drop-down list in the BO Type column. This is the type of business object definition that this port supports.
5. Click Apply to save changes.

Note: In some cases, not all ports in a collaboration object are needed; in this situation you must configure the collaboration logic to prevent the execution of service calls to the unused port or ports.

Because WebSphere InterChange Server requires that all collaboration ports be bound, you must also bind the unused port or ports to a Port connector. A *Port connector* is a generic connector definition that is used to close an unused port. Note that the Port connector must be used in conjunction with the correct collaboration logic; any service call sent to a port bound to a Port connector blocks the collaboration thread.

Changing a port name

To change the name of a port, you must delete and re-create the port using the new name; you cannot simply edit its name. Do the following to rename a port:

1. Select the port in the table on the Ports and Triggering Events tab.
2. Click Delete Port.
3. Follow the instructions in “Creating a port” on page 95 to create a port with the new name.

Table 32 summarizes what happens when you delete and re-create a port.

Table 32. Result of changing a port name

What Process Designer does	What you must do
The system-generated template variable that uses the port name changes.	If you have code that uses the variable declared with the old port name, change the variable name in the code. Find all action nodes and service calls in which the variable declared with the old port name appears. The compiler catches any remaining incorrect names.
The assignment of flow triggers (triggering events or triggering access calls) is deleted.	Reassign the flow triggers. See “Compiling a collaboration template” on page 101.

Defining scenarios

A *scenario* is the collaboration template code that handles a particular incoming business object or set of business object. This business object can represent an event (from a connector) or an access call (from an access client). You can think of a scenario as an event-handling method of your collaboration template class. Activity diagrams contain the code specifying how to handle the event.

About scenarios

You use scenarios to partition the business problem that a collaboration solves. You can group all the logic of the collaboration into a single scenario or you can create several scenarios, each dealing with one aspect of the problem. Grouping all collaboration logic into a single scenario is analogous to a program that contains all

of its logic in a main() function, while using multiple scenarios is analogous to a program that is structured into separate functions.

You typically name scenarios according to the function they perform. When a collaboration contains multiple scenarios, each of which handles one type of business object, consider naming each scenario according to the business object that it handles. For example, if the collaboration handles one type of business object with different possible verbs, you can develop Create, Update, and Delete scenarios. If the collaboration handles different types of business objects, you can develop a scenario for each business object definition.

A scenario handles only one triggering flow (triggering event or triggering access call) with each execution. However, the same scenario can potentially handle a set of possible triggering flows. For example, the same scenario can handle a Create, Update, or Delete flow.

In general, when identical logic handles different types of business object, it is more efficient to use a single scenario for those business objects. This eliminates the need to test and debug multiple pieces of code.

Note: A scenario cannot pass control to another scenario in the same collaboration. If your preliminary plans for partitioning the collaboration logic indicate that one scenario must call another, put all of the collaboration logic into the same scenario. Within the scenario, design is very flexible. Alternatively, you can create a collaboration group, dividing the logic among collaborations in the group.

Creating a scenario

Perform the following steps to create a new scenario:

1. From within Process Designer, click Template —> New Scenario. The Create Scenario dialog box is displayed.
2. Type the scenario's name in the Scenario Name field.
The name is a string that can contain alphanumeric characters and underscores. If the scenario handles events with a particular verb, it can be useful to include the verb in the scenario name.
3. Optionally, enter a description in the Description field.
4. Click OK. In the template tree view, the name of the new scenario displays in the scenario tree. In addition, the diagram editor opens in the main window.
5. You must assign at least one flow trigger to a scenario. Failure to assign the flow trigger causes a runtime error. For instructions on assigning triggering events to your new scenario definition, see "Assigning triggering events to scenarios."

Assigning triggering events to scenarios

You assign a triggering event to a scenario in the Ports and Triggering Events table of the Ports and Triggering Events tab. For each scenario that you create, you must assign its triggering event. The triggering event is represented by a business object and a verb.

Note: The general term for the incoming business object and verb that a scenario receives is a "flow trigger". If the flow trigger originates from a connector, it is called a triggering event. If the flow trigger originates from an access client, it is a triggering access method. The Ports and Triggering Events tab

enables you to assign a flow trigger to a scenario, regardless of whether it is a triggering event or a triggering access method. This section uses the terms “triggering event” and “event” because flow triggers received from connectors are by far the most common.

A collaboration’s port definitions specify the types of business objects that the collaboration can send and receive. After defining the collaboration’s ports and scenarios, you must specify:

- The port or ports through which triggering events enter and exit
In the Ports and Triggering Events tab, choose the row in the table that corresponds to the port name through which the triggering event enters and the business object name that represents the event.
- The object that triggers the collaboration’s execution
The flow trigger is represented by the port business object and a verb (*business-object.verb* combinations). In the row of the port and business object for which you are defining flow triggers, you specify the flow trigger by choosing its verb.
- The scenario that handles each flow trigger

Figure 34 illustrates these associations in a collaboration template whose port, From, supports business object type Widget. The Create scenario handles triggering event Widget.Create and the Delete scenario handles triggering event Widget.Delete.

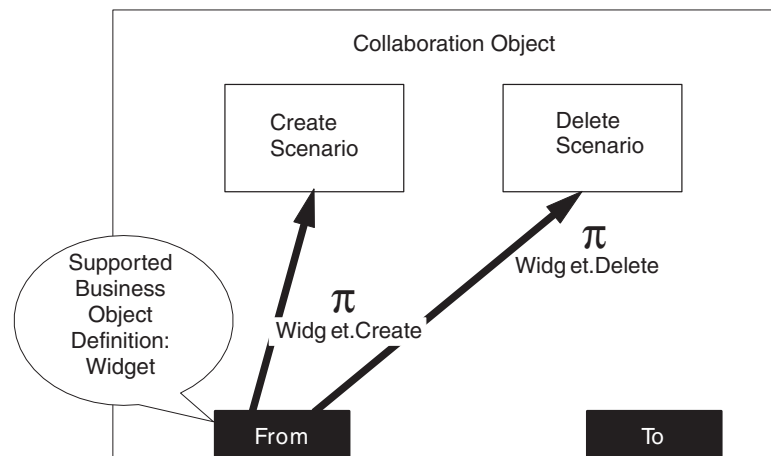


Figure 34. Relationship of port, triggering event, and scenario

Specify the scenario for each flow trigger as follows:

1. Ensure the Template Definitions window is open and that the Ports and Triggering Events tab is displayed.
2. In the Ports and Triggering Events table, locate the row that represents the port from which the flow trigger arrives, and the business object that represents the flow trigger.
3. In that row, click the drop-down list in the Create column. The list contains all of the scenarios defined for the template; select the scenario you want.
4. Repeat step 2 and step 3 for each port, business object, and verb whose flow trigger you want to assign.
5. After you have finished assigning triggering events, click Apply to save the assignments.

Defining scenario variables

After the scenario has been created, you can add scenario-specific variables in the Scenario Definitions dialog box (see Figure 35).

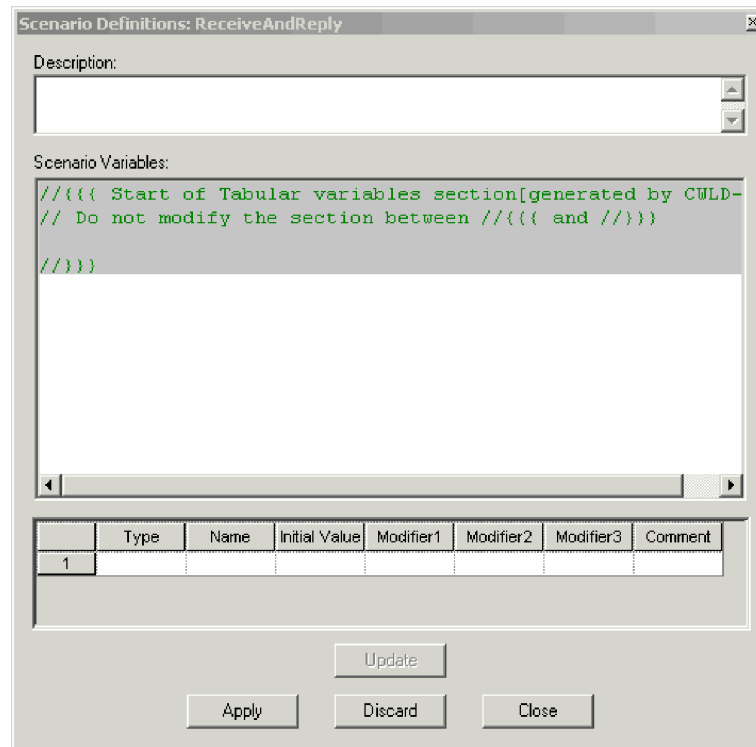


Figure 35. Scenario Definitions dialog box

Scenario variables are collaboration variables whose scope is all actions and links in a single scenario. (They are comparable to class variables in the Java programming language.) You can set scenario variables at any time during the collaboration template development process.

To add variables to the scenario definition, do the following:

1. Open the Scenario Definitions dialog box by doing one of the following:
 - Select a scenario in the template tree view and click Template → Open Scenario Definition.
 - Select a scenario in the template tree view and right-click to bring up the context menu. From the context menu, click Open Scenario Definition.
 - From an activity diagram in the diagram editor, right-click to bring up the context menu. From the context menu, click Open Scenario Definition.
2. Right-click the left heading cell of the variable table, and then click Add from the context menu. A new row appears in the table.

Note: You can also add a new row by clicking on the last row currently in the table.

3. Use the drop-down list in the Type column to specify the type of variable you want to declare.
4. Specify the variable's name in the Name column.
5. Specify the variable's initial value in the Initial Value column.

Note: String values must be entered in quotation marks (for example, the string Yes must be entered as "Yes.")

6. Specify any modifiers you want to apply to the variable (for example, transient, private, protected) in the Modifier1, Modifier2, and Modifier3 columns. Note that you do not have to specify a modifier in all three columns.

Note: Do *not* include the keywords public and static in the declaration of a scenario variable.

7. Click Update to add the new variable to the list of declarations at the top of the tab, and then click Apply to save the changes.

Special considerations for scenario variables in a long-lived business processes

Scenario variables do not persist automatically as part of the event flow context of a long-lived business process. If you want to use scenario variables within a long-lived business process collaboration, you must manually set the variable to null before the service call, and then re-initialize the variable after the service call completes. These tasks are done in the action node that makes the service call.

In the following example, a scenario variable called poolName is set to null in the action node before the service call takes place:

```
String poolName;  
poolName = null;
```

After the service call completes, poolName is re-initialized in the action node, as follows:

```
poolName = getConfigProperty("Pool_A");
```

Deleting a scenario

You can use Process Designer to delete scenarios. Deletion of scenarios cannot be undone.

To delete a scenario definition, do the following:

1. From the template tree view, select the scenario you want to delete.
2. Click Template → Delete Scenario. A dialog box appears to confirm the deletion.
3. Click Yes to delete the scenario.

Creating an activity diagram

Each scenario must have an activity diagram. An *activity diagram* uses Unified Modified Language (UML) to model the business process of the collaboration. UML represents the steps and decisions of the business process. You create an activity diagram in the diagram editor of Process Designer.

For detailed instructions on creating an activity diagram, see Chapter 5, "Using activity diagrams," on page 109.

Creating the message file

Part of the process of creating a collaboration template is defining its messages. The collaboration runtime environment uses the contents of the message file as the text for logging, tracing, and exception messages.

Process Designer provides the Template Messages view to facilitate message creation. The message text specified is stored as part of the collaboration template. When you compile and deploy the template, Process Designer extracts the message content and creates or updates the message file for runtime use.

For detailed instructions on creating a message file, see Chapter 10, “Creating a message file,” on page 231.

Compiling a collaboration template

The final task required to build a collaboration template is compiling the template. After you define the template properties, scenarios, activity diagrams, and message file, you must compile the collaboration template. The following files are created during compilation:

- Java source file (*CollaborationName.java*)
- Executable file (*CollaborationName.class*)
- Message text file (*CollaborationName.txt*)

After the template is compiled in Process Designer, these files are created in your Integration Component Library user project in System Manager. (For exact locations, see “Compiling a collaboration template” on page 6.)

Process Designer offers two ways to compile collaboration templates:

- “Compiling a single template”
- “Compiling multiple collaboration templates”

Compiling a single template

There are several ways to initiate compilation of a single collaboration template:

- From within Process Designer, click File → Compile.
- Select the template’s name in the template tree view and right-click to bring up the context menu, and then click Compile Template.
- Use the Ctrl+F7 keyboard shortcut.

If the Compile Output window is not already open, Process Designer opens it at the bottom of the main window to display compilation messages.

If an error occurs during compilation, do the following:

1. Trace the error by double-clicking the error message in the output window.
The activity diagram whose code generated the compilation error appears, with the faulty node selected.
2. Fix the problem and recompile. Repeat this process until you get the message:
Code Generator: Code generation succeeded.

Compiling multiple collaboration templates

The Process Designer File menu includes a Compile All menu option that enables you to compile all (or a subset) of the collaboration templates in your Integration Component Library user project. Perform the following steps to compile multiple templates:

1. If you have a template open in Process Designer, close it now.

2. Click File —> Compile All. The Compile All Templates dialog box opens. It displays a grid of all templates in the user project. By default, all templates are selected for compilation.
3. Clear the checkboxes next to any templates you do not want to compile.
4. Click Continue.
5. When you are prompted to confirm the compilation, click Yes.

Converting templates

Process Designer provides the following conversion functionality:

- Import—Process Designer can import Business Process Execution Language (BPEL) and Unified Modeling Language (UML in XMI 1.1) files for use as a collaboration template. See “Importing files.”
- Export—Process Designer can export your collaboration template to BPEL or UML (in XMI 1.1) format. See “Exporting a collaboration template.”
- Update—Process Designer can convert older collaboration templates for use with the current version of WebSphere InterChange Server. See “Updating templates from previous releases” on page 103.

Importing files

Process Designer can import BPEL and UML (in XMI 1.1) files for use in a collaboration template. Use the information in these files to create a new template definition.

Perform the following tasks to create a new collaboration template based on existing BPEL or UML (in XMI 1.1) files:

1. Ensure Process Designer is open.
2. Click File —> Import. The Process Designer Importer opens.
3. Select the file type you want to import, and then click Next.
4. Select the location of the BPEL or UML source file or files.
5. Select the file or files you want to import.

Note: If you are planning to use BPEL files, you must import all three of the `.bpel`, `.wsdl`, and `.bpelGUI.xml` files. Use the Ctrl key to select all three files for import.

6. Click Next to begin the import process. After the import is complete, the New Template dialog box opens.
7. Select the name of the user project the template belongs to in the Project field.
8. Type the name of the template you are creating in the Template Name field. A template name can include alphabetic characters, numbers, and underscores.
9. Click OK. Process Designer creates the new collaboration template and populates it with all of the information contained in the source BPEL or UML files.

Exporting a collaboration template

You can export your collaboration template into BPEL or UML (in XMI 1.1) format for use in other applications. When a WebSphere InterChange Server collaboration template is exported to BPEL format, the following files are created:

- `*.bpel`—This file contains the main template information.
- `*.wsdl`—This file contains information about the external interface.

- *.bpelGUI.xml—This file contains information about the graphical representation of activity diagrams. It is used in situations where BPEL files are imported back into WebSphere InterChange Server.

When a collaboration template is exported to UML (in XMI 1.1), a *.xmi file is created.

Perform the following steps to export a WebSphere InterChange Server collaboration template:

1. Ensure that Process Designer is open and that your collaboration template has been saved and has compiled without error.
2. Click File —> Export. The Process Designer Exporter opens.
3. Select the format to which you want to export your template, and then click Next.
4. Select the location in which you want to save the exported template file or files.
5. In the File Name field, specify the name for the exported template file. If you are exporting to BPEL, do *not* specify a file extension in the File Name field.
6. Click Next to begin the export process. The Process Designer Exporter dialog shows the progress of the conversion.
7. Click Close when the export process has finished.

Updating templates from previous releases

If you have a collaboration template from a previous release, you can use it with the latest version of WebSphere InterChange Server. The process for importing and updating these templates varies depending on whether the old template is stored in ICS or on a local file system.

If your existing template is stored in ICS, use System Manager to import it into a project as follows:

1. Ensure System Manager is open and is connected to the ICS where the old template is stored.
2. Locate the appropriate project folder in the Integration Component Library and right-click it.
3. From the project folder's context menu, click Import components from server —> Collaboration templates. The Import Components dialog box is displayed.
4. Select the collaboration template you want to import, and then click Finish.

The template is automatically updated for use with the latest version of WebSphere InterChange Server. After the template has been imported and updated, you can use Process Designer to edit it as needed.

If you want to use an existing collaboration template that is not stored in ICS, you must open it in Process Designer with the File —> Open —> From file command. If the template is not compatible with the latest version of WebSphere InterChange Server, Process Designer prompts you to update it. Confirm the update; Process Designer converts the existing template into a usable format.

After the conversion completes, it is good practice to visually inspect the new activity diagram for correctness. When you are satisfied with the template, save it to keep the changes.

Considerations when modifying templates.

Collaborations may require changes over time as businesses grow and change. Some of the changes will be to the Java code and others will be to the template definition itself.

You can change the Java code that supports a collaboration without adversely affecting existing collaborations; they will use the changed code on the next instantiation of the collaboration.

If you change the definition of a template, you **must** recreate any collaboration objects based on the template. The changes that require the collaboration object include (but are not limited to):

- Defining new attributes
- Removing existing attributes
- Removing ports
- Adding ports
- Changing triggering objects
- Removing triggering objects
- Adding triggering objects
- Adding business objects
- Removing business objects
- Changing business objects
- Adding new scenarios
- Removing scenarios

When you recreate the collaboration object, compile and test it as described in “Testing a collaboration” on page 107.

Special considerations for collaborations that use the SEND_EMAIL and INFORMATIONAL_EXCEPTIONS properties

Many collaboration templates based on the CollaborationFoundation template or BaseCollaboration template contain the SEND_EMAIL and INFORMATIONAL_EXCEPTIONS configuration properties (typically in the action nodes under the Main scenario). These properties determine how a collaboration handles exceptions; values can include specific exception message numbers.

In previous releases, the following code was used to obtain exception messages and parse them for the message number:

```
//Get exception message
sMessage=currentException.getMessage();

//Get exception type
sExceptionType=currentException.getType();

//Determine whether to send email. If SEND_EMAIL is set to none,
//then never send email. If SEND_EMAIL is set to all, then always
//send email. Else, make sure the message number is included in
//the comma-delimited list of SEND_EMAIL values.

StringTokenizer emailtoken=new StringTokenizer(sSendEmail, ",");
bBranch=false;

if (sSendEmail.trim().toLowerCase().equals("all"))
{
    bBranch=true;
}
```



```

else while (emailtoken.hasMoreTokens())
{
    sToken=emailtoken.nextToken();
    sToken=sToken.trim().toLowerCase();
    sToken="Error"+sToken;
    if (sMessage.lastIndexOf(sToken)>-1)
    {
        bBranch=true;
        break;
    }
}
}

```

The `getMsgNumber()` method has been added to the `CollaborationExceptions` class; it returns exception message numbers that are in turn used by the `SEND_EMAIL` and `INFORMATIONAL_EXCEPTIONS` properties in determining how an exception is handled. As a result, it can be necessary to manually update code in an existing collaboration template to take advantage of the `getMsgNumber()` method.

Determine whether the template you are converting contains the `SEND_EMAIL` or `INFORMATIONAL_EXCEPTIONS` configuration properties. If the template contains one or both of these properties, do the following:

1. Evaluate each scenario that contains `SEND_EMAIL` or `INFORMATIONAL_EXCEPTIONS` to identify any customized code (code other than that provided in the collaboration templates shipped with the product) in the action nodes. If the scenario does *not* contain any customized code, continue with Step 2.

If the scenario contains customized code to evaluate an exception message string for a message number, remove that code and then continue with Step 2.

2. Add the following code to any action node that returns exception messages and parses for exception message numbers:

```

//Get Exception Number
iMessageNum=currentException.getMsgNumber();
sMessageNum=String.valueOf(iMessageNum);

```

3. In the same action node, replace the existing while loop that returns the message number by parsing for the string `Error` with new code, as follows:

Existing code to replace

```

else while (emailtoken.hasMoreTokens())
{
    sToken=emailtoken.nextToken();
    sToken=sToken.trim().toLowerCase();
    sToken="Error"+sToken;
    if (sMessage.lastIndexOf(sToken)>-1)
    {
        bBranch=true;
        break;
    }
}

```

New code

```

else while (emailtoken.hasMoreTokens())
{
    sToken=emailtoken.nextToken();
    sToken=sToken.trim().toLowerCase();

    if (sToken.equals(sMessageNum))
    {
        bBranch=true;
        break;
    }
}

```

- If the action node uses SEND_EMAIL to determine collaboration exception handling, add the following code to support the ability to specify a range of exception message numbers. This code is nested inside the while loop you modified in Step 3 on page 105; the example below shows the whole while loop so you can locate the insertion point for the new code.

```

else while (emailtoken.hasMoreTokens())
{
    sToken = emailtoken.nextToken();
    sToken = sToken.trim().toLowerCase();

    trace (3,"***** Current sToken value is: "+sToken);

    String sSeparator="-";

    if (sToken.lastIndexOf(sSeparator)>-1)
    {
        int pos = sToken.indexOf(sSeparator);

        sLowerRange = sToken.substring(0,pos);
        sLowerRange = sLowerRange.trim();
        iTemp = new Integer(sLowerRange);
        iLowerRange = (((Integer)iTemp).intValue());

        sUpperRange = sToken.substring(pos+1,sToken.length());
        sUpperRange = sUpperRange.trim();
        iTemp = new Integer(sUpperRange);
        iUpperRange = (((Integer)iTemp).intValue());

        trace(3," ***** iLowerRange value is -" +iLowerRange
            +" *****iUpperRange value is -"
            +iUpperRange +" *****");

        if ((iMessageNum>=iLowerRange) && (iMessageNum<=iUpperRange))
        {
            bBranch = true;
            trace(3," ***** The current exception falls within
                this range *****");
            break;
        }
    }
    else
    {
        if (sToken.equals(sMessageNum))
        {
            bBranch=true;
            break;
        }
    }
}

```

- For each scenario that contains the SEND_EMAIL or INFORMATION_EXCEPTIONS properties, add the following scenario variables in the Scenario Definitions dialog box. For more information on adding scenario variables, see “Defining scenario variables” on page 99.

Table 33. Scenario variables to add during template conversion

Variable type	Variable name	Initial value
String	sMessageNum	null
String	sLowerRange	null
String	sUpperRange	null
int	iMessageNum	0
int	iLowerRange	0

Table 33. Scenario variables to add during template conversion (continued)

Variable type	Variable name	Initial value
int	iUpperRange	0

6. Recompile your template and test it to ensure the changes are correct. See “Compiling a collaboration template” on page 101 and “Testing a collaboration” for more information on these tasks.

Deleting a collaboration template

Important

Do not delete a collaboration template that has collaboration objects associated with it, unless you plan to delete those collaboration objects. Deleting a collaboration template renders all objects built from that template unusable. (For instructions on deleting collaboration objects, and then deleting the collaboration template in System Manager, see the *Implementation Guide for WebSphere InterChange Server*.)

Use Process Designer to delete a collaboration template that does not have a collaboration object built from it. To delete a template, do the following:

1. Open Process Designer and ensure that System Manager is running.
2. Click File —> Delete. Process Designer displays the Delete Template from Project ‘ProjectName’ dialog box.
3. From the Project drop-down list, select the name of the project that contains the template you want to delete.
4. From the list of collaboration templates, select the name of the template you want to delete, and then click OK.
5. The tool prompts you to confirm the deletion. Click Yes.

Testing a collaboration

After you have built and successfully compiled a collaboration template, you can test its design. To verify that your collaboration works as planned, you must create a collaboration object and use the Test Connector tool to test the collaboration object’s functionality.

The Test Connector, which is part of the WebSphere InterChange Server Testing Environment, simulates an actual connector. Use the Test Connector to send events and responses to collaborations. It enables you to set up business objects and triggering events that test the functionality of a collaboration.

If the collaboration you are testing has a port to one connector, then you open one instance of the Test Connector. If the collaboration uses an incoming port from one connector and another port to a different connector, then you open two instances of the Test Connector, one for each connector.

From the Test Connector menus, you designate the configuration file and the definition of the connector to be emulated. You set up values for the selected business objects, then send and receive the business object.

For detailed information on using the Integrated Testing Environment and Test Connector, see the *Implementation Guide for WebSphere InterChange Server*.

Debugging a collaboration

The Collaboration Debugger tool enables you to step through all of the business logic in a collaboration template to ensure it works as designed. Collaboration Debugger provides functionality for setting multiple breakpoints in a scenario; the debugging process can stop at each breakpoint to enable you to collect relevant data about the triggering event and its flow through the scenario.

Collaboration Debugger is available from within System Manager (as a perspective) and the Integrated Test Environment. For more information on using Collaboration Debugger, see the *Implementation Guide for WebSphere InterChange Server*.

Chapter 5. Using activity diagrams

This chapter shows how to use Process Designer to edit an activity diagram. An *activity diagram* defines the control flow of that particular part of a collaboration; it is created automatically when you create a scenario. The diagram is a set of steps that execute in a specified order. An activity diagram contains symbols that specify the steps, the order of the steps, and the logic that determines how they execute.

For information on laying out and viewing the workspace, refer to Chapter 8, “Workspace and layout options,” on page 183.

To edit an activity diagram for a scenario, do the following tasks:

1. Display the diagram editor in the Working Area.

You can bring up the diagram editor in either of the following ways:

- Select a scenario name or diagram name in the template tree view and choose the Open Diagram option from the Template pull-down menu.
- Select a scenario name or diagram name in the template tree view and right-click to bring up the context menu. From the context menu, choose Open Diagram.
- Double-click the scenario name.

Process Designer displays the diagram editor with the activity diagram for the selected scenario.

2. Edit the activity diagram in the diagram editor, using the symbols provided in the Symbols toolbar.
3. Save the activity diagram with the Save option of the File menu, or use the shortcut key combination Ctrl+S.

Using the diagram editor functionality

You can access the diagram editor’s functionality in either of the following ways:

- Toolbars
- Mouse movements on symbols in the activity diagram

Accessing diagram editor functionality: Process Designer menus

Process Designer has pull-down menus from which you initiate many of the diagram-related operations. There are keystroke shortcuts and context menus for some of these functions. For detailed information about these menus, see “Process Designer menus” on page 20.

Accessing diagram editor functionality: Mouse movements

The diagram editor recognizes the following mouse movements:

- Click the left mouse button to select a component or symbol in an activity diagram.

The diagram editor surrounds the symbol with grey anchor squares, called the *selection border*, to indicate that the symbol is selected. To deselect the symbol, click elsewhere in the workspace.

- Click the right mouse button to select and bring up a context menu for the symbol.
The context menu contains the following options:
 - Properties—Displays the appropriate Symbol Properties dialog for the selected symbol. Same as the Properties option of the Edit menu. For more information, see the description of the Properties option in “Functions of the Edit menu” on page 21.
 - Font—Controls the font in which the symbol text displays. Same as the Font option of the Edit menu. For more information, see the description of the Font option in “Functions of the Edit menu” on page 21.
- Click the right mouse button anywhere within an activity diagram (but not on a specific symbol) to navigate to the activity diagram’s parent diagram.

Activity diagram symbols

An activity diagram uses symbols to represent the steps of execution. This section provides the following information about the symbols of an activity diagram:

- What types of symbols exist?
- How do symbols compare with those in a flow chart?
- What are the properties of every symbol?

Introduction to the symbols

Figure 36 shows the symbols of an activity diagram and their associated buttons in the Symbols toolbar. This toolbar becomes active when the diagram editor displays in the Working Area.

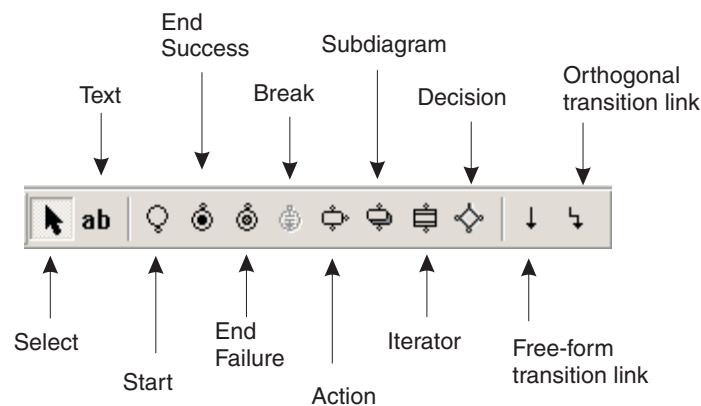


Figure 36. Symbols toolbar

Activity diagrams contain three main types of symbols: nodes, transition links, and service calls. In addition, they contain start and end symbols.

Start and End symbols

When an activity diagram is created, a Start symbol is automatically placed in the diagram. This symbol represents the beginning of the flow; each activity diagram must have a Start symbol.

The Start symbol can be used to initialize a correlation attribute. For more information, see “Using a correlation attribute” on page 132.



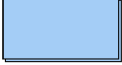

Process Designer provides two end symbols for activity diagrams: End Success and End Failure. Each execution path in an activity diagram must end with one of these symbols (with the exception of an iterator activity diagram that ends with a break symbol). For more information on using the end symbols, see “Terminating the execution path” on page 143.

Node symbols

A *node* is a symbol that represents a step in a collaboration. There are four types of nodes: *actions*, *decisions*, *subdiagrams*, and *iterators*. Each node is represented by a unique symbol in the Symbols toolbar (see Figure 36 on page 110).

Table 34 illustrates the symbol placed in the activity diagram for each type of node.

Table 34. Node symbols



Node type	Symbol in activity diagram	For more information
Action		“Action nodes” on page 112
Decision		“Decision nodes” on page 120
Subdiagram		“Subdiagrams” on page 134
Iterator		“Iterators” on page 139

Transition link symbols

A *transition link* represents control flow between nodes. Because the flow of a diagram is from top to bottom, a transition link is always oriented vertically. If multiple paths are available from a node, the transition link must be used with a decision node. Logic in the decision node determines which path is taken.

The diagram editor can represent a transition link in one of two ways: free-form links and orthogonal links. Table 35 shows the activity diagram symbol that represents each type of transition link.

Table 35. Transition-link symbols

Transition link type	Symbol in activity diagram	For more information
Free-form transition link		“Transition Links” on page 116
Orthogonal transition link		“Transition Links” on page 116

Service call symbol

A *service call* represents a request to or response from an outside entity, through a port. It is always oriented horizontally. A service call is attached to an action node. By default, the label of a service call describes the service call type. Service calls can be one of the following:

- Synchronous Service Call
- Asynchronous Outbound Service Call
- Asynchronous Inbound Service Call

Note that the Symbols toolbar does not contain a symbol for service calls. Service call functionality is available through the context menu that appears when you right-click an action node.

For more information on the types of service calls and how to include them in an activity diagram, see “Service calls” on page 126.

Properties of diagram symbols

A symbol in an activity diagram can have the properties shown in Table 36.

Table 36. Properties of a symbol

Symbol property	Description
A unique identifier (UID)	Every symbol in an activity diagram has a unique identifier (UID). You can choose whether or not to display the UID in your diagram. Although you can assign your own labels to symbols, your own label does not replace the UID. The UID identifies the symbol in compilation and tracing messages. You can choose whether to display the UIDs with the View UIDs option of the View pull-down menu.
An optional label	The label serves as a descriptive name that makes the activity diagram more readable (when labels display). You can choose whether to display the labels with the View Labels option of the View pull-down menu.
An optional description	The description is a comment.
Type-specific properties	Some symbols, such as action nodes, have an associated code fragment.

You can edit properties of most symbols. Bring up the Symbol Properties dialog in one of these ways:

- Right-click a symbol in an activity diagram to bring up its context menu, from which you select Properties.
- Select a symbol in an activity diagram, then select Properties from the Edit pull-down menu.
- Double-click a symbol in an activity diagram to open its Symbol Properties dialog.
- Use the shortcut combination Ctrl + Enter.

Action nodes

An action node (often called simply an *action*) represents a step in a collaboration. It is the basic building block of collaboration logic. The breakdown of the collaboration’s logic into action nodes is completely up to you. You can write many lines of complex code in a single action or divide the logic into numerous individual actions. Breaking a collaboration’s logic into action nodes is analogous to developing program code. You can write a program with a short main routine

that invokes a series of subroutines or method calls to carry out the program function. Or, you can write a longer main routine that includes all program logic inline.

Adding an action to a diagram

To add an action node to an activity diagram:

1. In the Symbols toolbar, click the Action button.
2. Click in the workspace to place the Action symbol.

Note: An action node can make a service call if you attach the Service Call symbol to the action. For information on service calls, refer to “Service calls” on page 126.

Defining action node properties

After the action node appears in the activity diagram, use the Action Properties dialog box to define any of the following properties for the node:

- Label—Provides a label for the action node. Using descriptive text instead of the default UID makes the diagram easier to read and use. This property is optional.
- Description—Provides a description of the action node’s purpose. This property is optional.
- Code fragment—Defines what the action node does. For more information, see “Adding activity definitions to an action node” on page 114.

Open the Action Properties dialog box by doing one of the following:

- Double-click on the selected action node.
- Right-click on the action node to bring up the context menu, then choose Properties.
- Select the action node; then choose Properties from the Edit pull-down menu.
- Select the action node; then use the keyboard shortcut Ctrl + Enter.

The Action Properties dialog displays with the name of the action node at the top of the dialog. This name has the following format:

Action_UID

where *UID* specifies the unique identifier for the action node. Figure 37 on page 114 shows the Action Properties dialog box.

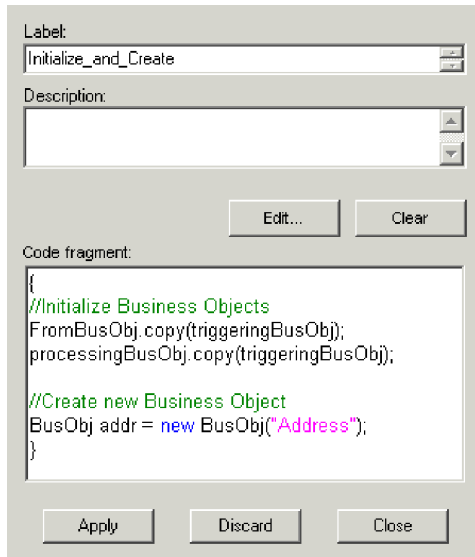


Figure 37. Action Properties dialog

Adding activity definitions to an action node

Action nodes contain activity definitions. Activity definitions (also called *code fragments*) consist of calls to the collaboration API or other Java code, and can contain operations such as the following:

- Getting and setting the attribute values in business objects
- Checking the verb in an incoming event
- Comparing attribute values to constants or to other attribute values
- Setting up business object variables to use in service calls
- Logging messages

Activity definitions can contain any of the Java programming language constructs.

Activity definitions can be added to action nodes in two ways:

- By typing Java code directly into the Code Fragment window of the Action Properties dialog box. See “Adding an activity definition in the Action Properties dialog box.”
- By using Activity Editor to add either graphical function blocks or traditional Java code. See “Adding an activity definition in Activity Editor” on page 115.

Adding an activity definition in the Action Properties dialog box

To add Java code directly to the Code Fragment window in the Action Properties dialog box, do the following:

1. Enable direct editing of the Code Fragment window as follows:
 - a. In Process Designer, click View —> Preferences. The User Preferences dialog box opens.
 - b. In the Diagram tab, select the Enable in-place editing for new action nodes and Enable in-place editing for existing action nodes options.
 - c. Click Apply, and then click OK.
2. Right-click the action node to display its context menu.

3. From the context menu, click Properties. The Action Properties dialog box is displayed.
4. Type your Java code into the Code Fragment window.
5. Click Apply to save your changes.

Adding an activity definition in Activity Editor

To add an activity definition using Activity Editor, do the following:

1. Right-click the action node to display its context menu.
2. From the context menu, click Properties. The Action Properties dialog box is displayed.
3. Click Edit to open the Activity Editor. By default, the Activity Editor displays the Graphical view.

Note: If you have already created an activity definition for this node, and if that definition contains custom Java code that is not recognized by the Graphical view, Activity Editor displays the Java view.

4. Add the activity definition. See “Activity definitions” on page 154 for more information.
5. Close the Activity Editor. The Action Properties dialog box is still open, and it now displays the code fragment associated with the activity definition.
6. Click Apply to save your changes.

Using Java relational operators in activity definitions

You can use the Java operators in activity definitions, particularly the relational and conditional operators along with the arithmetic operators.

All Java classes inherit from the base Object class. As such, every class has an equals() method available to it. Its signature is as follows:

```
public boolean equals(Object obj);
```

The equals() method is concerned with value equality. It compares the object calling the method and the object referenced by obj for equality, returning true if they have the same value and false if they do not. Note that this differs from the equality operators == and !=. These two equality operators determine if two references refer to the same object, regardless of the value of the object. This is an important distinction.

Table 37 summarizes the available equality operators. Each equality or relational operator yields a boolean value result.

Table 37. Java relational operators

Relational operator	Meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
!	unary operator, which inverts a boolean value.

Note that results of boolean expressions can be joined with the conditional AND (&&) and the conditional OR (||) symbols.

Table 38 summarizes the arithmetic operators, including the increment and decrement operators.

Table 38. Java arithmetic and assignment operators

Arithmetic operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	remainder
-	unary, for negation or inverting the sign of a number
++	increment operator
--	decrement operator
=	assignment operator

Using Web services in activity definitions

If you want to include a Web service in a collaboration template's activity definition, you must first export the Web service from the Integration Component Library (ICL) project in System Manager. Each method in the Web service is exported as a function block to Activity Editor, where it can be placed in the activity definition. For more information, see "Web services function blocks" on page 155.

Finding more information on activity definitions

For more information on coding logic for an action node, refer to the following table:

Table 39. Obtaining more information on code fragments

Additional resources	Location
Detailed description of Activity Editor, including examples of using function blocks	Chapter 6, "Using Activity Editor," on page 149
Examples of code fragments	Chapter 9, "Coding tips and examples," on page 193.
Reference pages for individual function blocks	Chapter 11 through Chapter 21
Reference pages for individual methods in the collaboration API	Chapter 22 through Chapter 29.

Transition Links

Transition links represent an activity diagram's control flow. They connect nodes in which activities occur, such as actions, decisions, subdiagrams, and iterators, and connect these nodes to start and completion symbols. Transition links can contain business object probes that monitor business object instance values.

Note: In activity diagrams, transition links do not represent data flow. Data passes from node to node when one node sets a variable and another accesses the variable. The activity diagram's data-flow mechanism is comparable to that of a class or program, in which code sets a variable that other code uses. This model differs from the one used by event-passing modeling tools, which show data moving along links.

Process Designer provides both orthogonal and free-form transition links. Use the orthogonal links whenever possible. Use the free-form links when you cannot get the desired shape from the orthogonal links. Right-click a link to see a context menu that shows the orthogonality of the link. Use this context menu to toggle a link between orthogonal and free form.

How many links can coexist?

Table 40 displays the number of incoming and outgoing links that different types of nodes can have.

Table 40. Permitted incoming and outgoing links by node type

Node type	Incoming links	Outgoing links
Action	Unlimited	One
Decision	One	Seven
Subdiagram or iterator	Unlimited	One

Creating a transition link

To create a transition link, the two symbols that you want to connect must be available on the workspace.

To add a transition link to an activity diagram:

1. In the Symbols toolbar, click the Transition Link button.
2. In the workspace, click the bottom edge of the symbol where you want the transition link to start.
3. Click the top edge of the symbol where you want the transition link to end.

Process Designer lets you place valid connections between symbols. It does not permit you to link two symbols for which a link is invalid. Rather than displaying an error message, Process Designer does not permit the invalid transition link to be made.

Process Designer indicates whether an attempt to place a transition link on a symbol is valid. When you position the mouse pointer (with the link) on the edge of a symbol, the mouse pointer changes to a plus sign within a circle if the connection is valid. You can then click and place the connection on the symbol. If the connection is not valid, the mouse pointer does not change to a plus sign and you cannot place the connection.

Canceling a link

Abort or cancel a transition link by pressing the Escape (ESC) key. Each press of the ESC key undoes the last segment of the connection line. Using the ESC key is the only way to cancel a connection attempt for which there is no valid symbol in the activity diagram.

For example, suppose you have placed two symbols, a start symbol and an end symbol, in a diagram. You then select a transition link and click the start symbol. The transition-link line segment appears, connected to the start symbol. However, there are no valid symbols or ports to which you can connect the transition-link line. At this point, pressing the ESC key is the only way to cancel the connection activity and continue the editing session.

Defining transition link properties

After the transition link appears in the activity diagram, you can define its properties in the Link Properties dialog (see Figure 38).

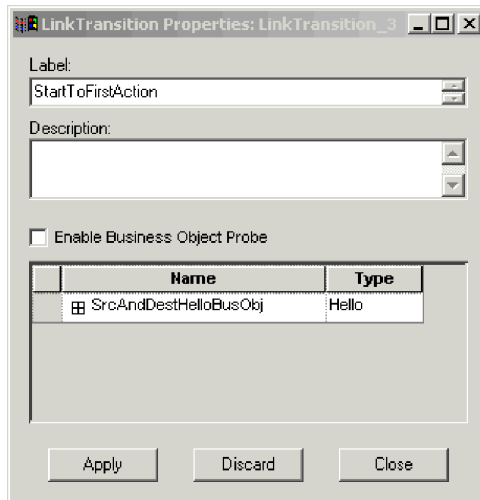


Figure 38. Link Transition Properties dialog box

The Link Properties dialog identifies the transition link with a name at the top of the dialog in the following format:

`LinkTransition_UID`

where *UID* is the unique identifier for the link.

The properties of a transition link include its label, definition, and whether it contains a business object probe.

To define transition link properties:

1. Display the Link Properties dialog.

You can display this dialog in any of the following ways:

- Double-click on the selected transition link.
- Right-click on the transition link to bring up the context menu, then choose Properties.
- Select the transition link; then choose Properties from the Edit pull-down menu.
- Select the transition link; then use the keyboard shortcut `Ctrl + Enter`.

The Link Properties dialog displays. Figure 38 shows a sample Link Properties dialog.

2. Optionally, specify the label and description for this transition link.
For more information on link labels, see “Labeling a link” on page 118.
3. Click Apply to save the link properties.

Labeling a link

Link labels can be instrumental in ensuring the readability of the activity diagram. Try to capture the decision logic in the same way that you would when labeling a decision node in a flow chart. Logically named link labels explain the scenario flow. For example:

- If two transition links branch based on the value of a configuration property called CONVERT_VERB, the labels might be DoConvert and DoNotConvert.
- If one transition link handles a successful service call and another link handles a service call exception, the labels might be Success and ServiceCallException.

To label a transition link, enter the text of the link label in the Label box of the Link Properties dialog. The label appears in the activity diagram exactly as it does in the text box. Use carriage returns in the text box to break a label into multiple lines so that it does not overlap other links.

Using business object probes

A business object probe monitors business object instance values during runtime. The probe is placed on a transition link during the creation of an activity diagram, and is activated or deactivated during runtime through System Manager's Collaboration Properties dialog box.

By default, a business object probe appears as a red square on the transition link in an activity diagram. In Figure 39, the Default branch link contains a business object probe.

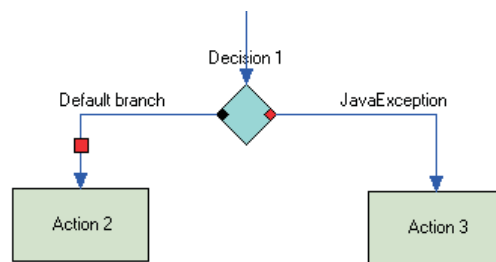


Figure 39. Transition link with business object probe

You can use a business object probe to monitor any business object specified in the Ports and Triggering Events tab of the Template Definitions window, with the following exceptions:

- Business object probes cannot be used on the incoming transition link for a decision node.
- Business object probes cannot be used on a service call link.

For each business object, you can choose the specific attributes you want to monitor. All of the instance values for these attributes are presented in a report provided by System Monitor.

Perform the following tasks to add a business object probe:

1. Ensure that Process Designer is open and that the activity diagram is displayed.
2. Right-click the transition link to which you want to add the business object probe.
3. From the context menu, click Properties. The Link Transition Properties dialog box opens.
4. Click the Enable Business Object Probe checkbox.
5. Click the plus sign (+) next to the business object you want to monitor to display that object's list of attributes.
6. Select the specific attributes you want to monitor.
7. Click Apply to save your changes.

8. During runtime, use System Manager to enable or disable the business object probe as needed.

Modifying a transition link

You can disconnect and reconnect transition links as well as modify the appearance of the transition-line segments.

- To disconnect and reconnect, select a link point—the end of a line—with the mouse and drag it in the desired direction. Disconnecting and reconnecting allows you to move a transition link.
- To create a new transition-line segment in an existing transition link, click the middle section of a line segment with the Ctrl key held down. The join between the two line segments is marked with a square.
- To remove a transition-line segment, click the line segment square with the Ctrl key held down.

Note: Modifying line segments does not apply to orthogonal links.

To ensure that transition-line segments have only right angles, hold down the Shift key while creating a new line segment.

Decision nodes

If you want one action to flow to the next regardless of conditions, a transition link is all that is necessary. If, however, you want to branch to more than one action based on a set of conditions, you need to include a decision node. In its most common usage, a decision node connects an action to all of its possible outcomes, including other actions, subdiagrams, and end symbols. Decision nodes can be used with the action, subdiagram, and iterator nodes. Do not place a decision node directly after a start symbol.

A decision node typically has at least two branches; the maximum number of branches is seven. Each branch has a condition associated with it that determines whether that branch is taken or not.

Important

When implementing a decision node, ensure that you define the conditions such that there will always be one that evaluates to true. If none of the conditions in your decision node evaluate to true, a runtime error occurs.

There are three types of branches in a decision node:

- Normal—A normal branch has a condition associated with it; if that condition is met, the branch is taken. You can have multiple normal branches. By default, normal branches are represented by a blue square.
- Exception—An exception branch has a specific exception type associated with it. The condition of an exception branch tests that the system variable `currentException` is equal to the exception type to which you set the branch. You can have multiple exception branches. By default, exception branches are represented by a red square.
- Default—The default branch is taken when none of the other branch conditions are true. Each decision node can have one (and only one) default branch. This branch is optional. By default, it is represented by a black square.

These branches are defined and their conditions set in the Decision Properties dialog box, as shown in Figure 40.

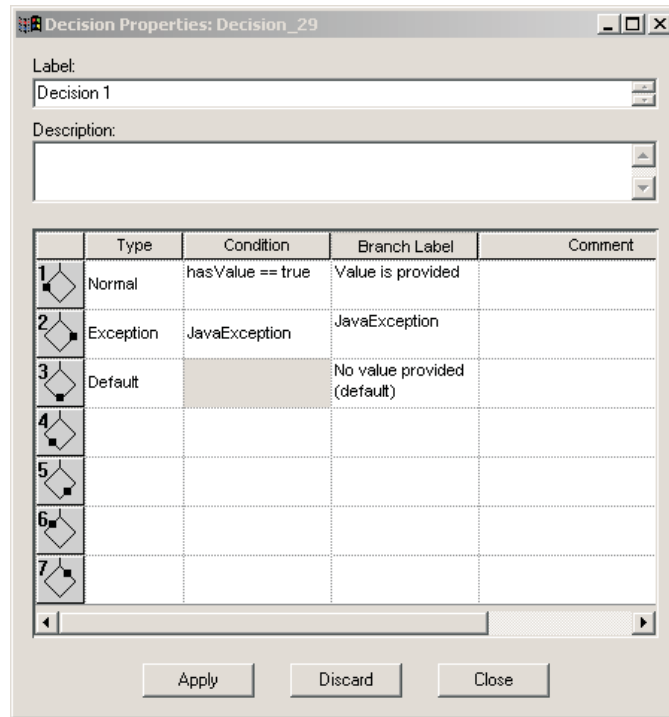


Figure 40. Decision Properties dialog box

Each defined branch of a decision node must have a transition link that connects it to its associated outcome (for example, an action node or an end symbol).

Figure 41 on page 122 illustrates a sample activity diagram with a decision node. In this example, the decision node has three branches. The normal branch shifts the flow to Action 2 if its condition evaluates to true. The exception branch shifts the flow to End Failure if a `JavaException` exception is thrown. The default branch shifts the flow to Action 3 if the condition of the normal branch evaluates to false and a `JavaException` is not encountered.

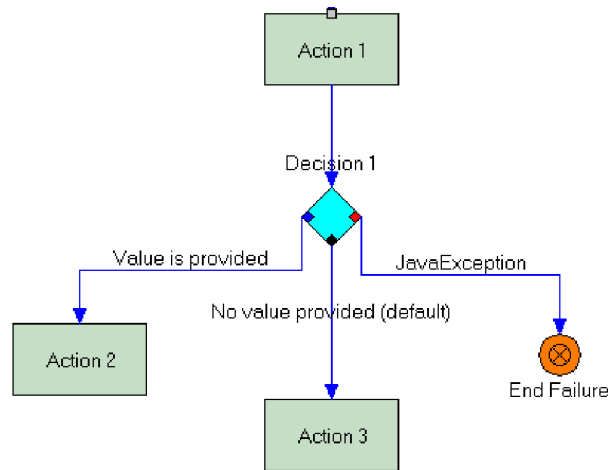


Figure 41. Activity diagram with a decision node

Perform the following steps to add a decision node to your activity diagram:

1. Ensure the diagram editor is open, and that you have already placed the symbol that is going to flow to the decision node. Decision nodes can be used by any action, subdiagram, or iterator node.
2. Click the Decision Node button in the Symbols toolbar.
3. In the diagram, position your cursor underneath the symbol that is going to use the decision node, and then click to place the node in the diagram.
4. Create a transition link between the decision node and the symbol that calls it. See “Creating a transition link” on page 117 for more information on creating transition links.

Defining a normal branch

Each normal branch requires a condition. These conditions are created with variables that you define in the collaboration template or the scenario. Before you can create a normal branch, you must define the necessary variable for the condition. See “Declaring and editing template variables (the Declarations tab)” on page 86 and “Defining scenario variables” on page 99 for more information.

Perform the following steps to define a normal branch in a decision node:

1. In the diagram editor, double-click the decision node symbol. The Decision Properties dialog box opens.
2. In the row for the branch you are creating, click the table cell in the Type column and select Normal from the drop-down list of branch types.
3. Right-click the table cell in the Condition column and select Condition Builder from the context menu.

Note: You can also type the condition directly into the Condition table cell instead of using the Condition Editor. The Condition Editor dialog box is displayed.

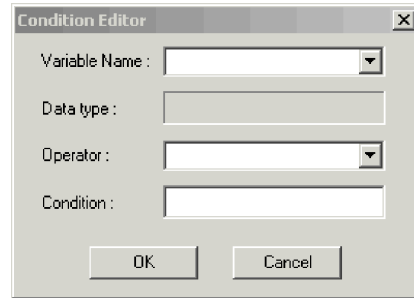


Figure 42. Condition Editor

4. In the Variable Name field, use the drop-down list to select the variable you want to evaluate for the condition. This list contains all of the collaboration variables you have defined for the scenario.
When you select a variable, the Data type field is automatically updated to include the variable type (for example, Boolean or String).
5. In the Operator field, use the drop-down list to select the appropriate operator to use for evaluating the variable. The list contains only those operators supported by the type of variable you are using.
6. In the Condition field, enter the value you want to use for the condition. (For example, if you have a Boolean variable named `hasValue`, you can set the condition to either `true` or `false`.)
7. Click Ok to close the Condition Editor and return to the Decision Properties dialog box.
8. Optionally, type a label for the branch in the Branch Label table cell. Labeling your branches can improve the readability of your activity diagram.
9. Optionally, type a description for the branch in the Comment table cell.
10. Click Apply to add the branch to the decision node. In the activity diagram, the decision node now contains a blue square to indicate the normal branch you just created.

After you add a normal branch, you must connect it to its associated result with a transition link.

Defining an exception branch

Perform the following steps to define an exception branch in a decision node:

1. In the diagram editor, double-click the decision node symbol. The Decision Properties dialog box opens.
2. In the row for the branch you are creating, click the table cell in the Type column and select Exception from the drop-down list of branch types.
3. Click the table cell in the Condition column and select the type of exception from the drop-down list of exception types.
4. Optionally, type a label for the branch in the Branch Label table cell. Labeling your branches can improve the readability of your activity diagram.
5. Optionally, type a description for the branch in the Comment table cell.
6. Click Apply to add the branch to the decision node. In the activity diagram, the decision node now contains a red square to indicate the exception branch you just created.

After you add an exception branch, you must connect it to its associated result with a transition link.

Defining a default branch

Each decision node can have only one default branch. Adding a default branch is optional.

Perform the following steps to add a default branch to your decision node:

1. In the diagram editor, double-click the decision node symbol. The Decision Properties dialog box opens.
2. In the row for the branch you are creating, click the table cell in the Type column and select Default from the drop-down list of branch types.
3. Optionally, type a label for the branch in the Branch Label table cell. Labeling your branches can improve the readability of your activity diagram.
4. Optionally, type a description for the branch in the Comment table cell.
5. Click Apply to add the branch to the decision node. In the activity diagram, the decision node now contains a black square to indicate the default branch you just created.

Note that you cannot specify a condition for the default branch. The condition is implicit; it evaluates to true when all of the conditions associated with the other branches evaluate to false.

After you add the default branch, you must connect it to its associated result with a transition link.

Combining an exception and a condition in branching logic

A branch can be normal or an exception, but not both. However, there are times when you might want to specify the execution path to take in response to two simultaneous conditions: an exception occurred and another condition is true. This combination is the equivalent of using an AND operator in a conditional expression.

For example, suppose you want to model these two conditions:

```
Exception == JavaException && hasValue == false  
Exception == JavaException && hasValue == true
```

To model such a construct, create two levels of decision nodes, putting an action node between them, as shown below:

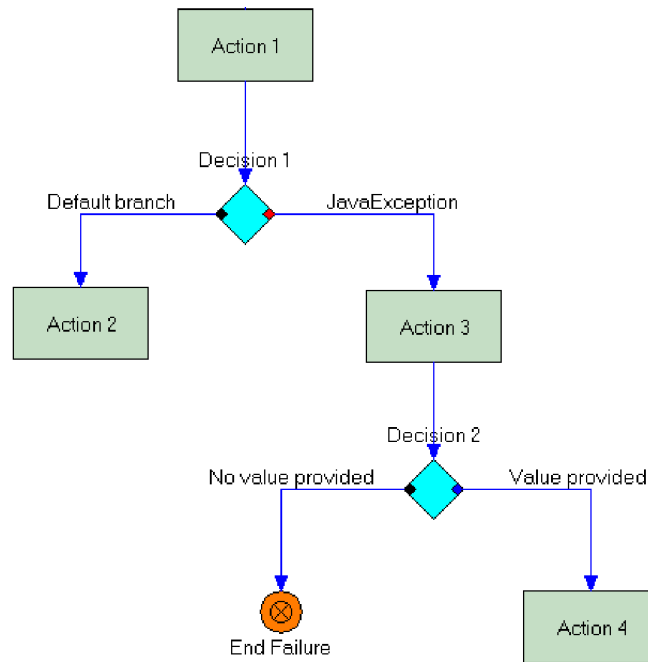


Figure 43. Combining an exception and a condition

Swapping decision node branches

You can swap branches in a decision node by essentially dragging and dropping the branches in the activity diagram. Process Designer automatically updates the Decision Properties dialog box to reflect the new location of each branch.

Perform the following steps to swap the location of two branches in a decision node:

1. In the diagram editor, locate the decision node whose branches you want to swap.
2. Click the transition link for one of the branches you want to swap.
3. While holding down the Alt key, drag the top of the transition link from its original branch to the new branch.
4. Release the Alt key. Process Designer swaps the two transition links and their associated conditions and updates the list of branches in the Decision Properties dialog box.

Service calls

An action node cannot, by itself, send a request to a connector or another collaboration. Instead, you must connect the action node to a service call. InterChange Server supports both synchronous and asynchronous service calls. The following is an example of a synchronous service call that uses a Retrieve request:

```
status = retrieve(business-object, port);
```

↑ ↑ ↑
Status Request Data, sent from
of type and returned to
completed the same variable
call

A service call always connects to an action. The action generates the service call and handles the service call's results on its outgoing transition links. An action and a service call work as a pair; the service call performs the action's remote input/output function.

Figure 44 illustrates the relationship between an action node and service call. The figure shows the order in which the collaboration runtime environment processes the action and its service call.

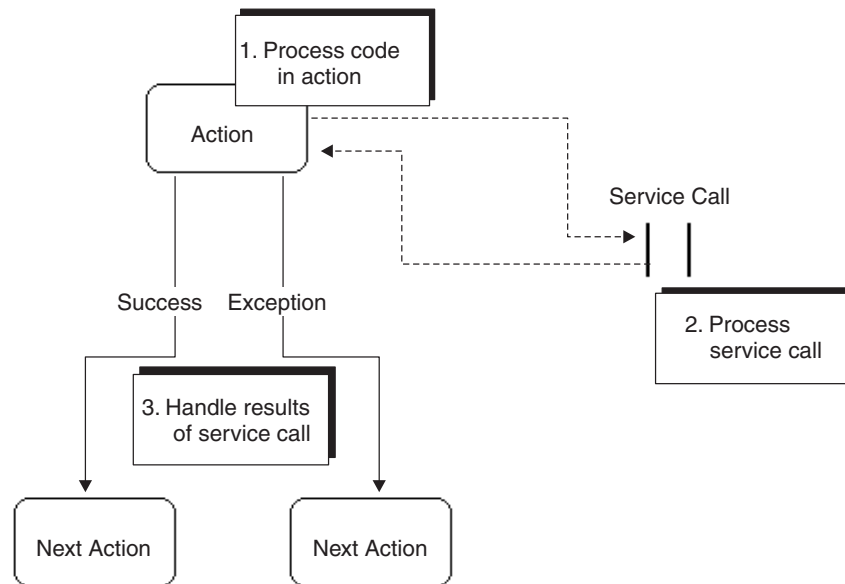


Figure 44. Relationship between action node and service call

Figure 44 shows that the collaboration runtime environment first processes the code in the action and then executes the service call. When the service call completes, the action node's outgoing transition links handle the result. When an action node generates a service call, it is good practice to include an outgoing exception link that checks for `ServiceCallException`.

Types of service calls

InterChange Server supports three types of service calls: synchronous, asynchronous outbound, and asynchronous inbound. The following sections describe each type of call.

Synchronous service call

This type of service call uses a synchronous request/response mechanism. The service call sends the request but does not complete until the response arrives and is processed.

Synchronous service calls support compensation. In addition, they support a timeout value for long-lived business processes.

By default, all service calls added to an activity diagram are synchronous. You can then change the type if required by your scenario.

Asynchronous outbound service call

An asynchronous outbound service call sends a request but does not expect or wait for a response before continuing its processing. A collaboration template must support long-lived business processes in order to use asynchronous outbound service calls.

If an asynchronous outbound service call has a port that is bound to a collaboration instead of a connector, the service call automatically becomes synchronous.

Asynchronous outbound service calls support compensation, but do not support a timeout value for long-lived business processes.

Asynchronous inbound service call

An asynchronous inbound service call waits to receive an incoming event based on a correlation attribute or set of correlation attributes that identify the event. It is used in conjunction with long-lived business processes. (See “Using a correlation attribute” on page 132 for more information.)

When an asynchronous inbound service call is created, it is given a timeout value; if the service call does not receive an incoming event before the timeout expires, the exception `TimeoutException` is raised.

Asynchronous inbound service calls are available only if the collaboration template supports long-lived business processes. You can enable this support at any time during the template development processes by clicking the Long Lived Business Process Support option on the Template Definitions General tab.

Asynchronous inbound service calls do not support compensation.

Creating a service call

Perform the following steps to add a service call to the activity diagram:

1. Ensure that the diagram editor is open.
2. In the workspace, right-click the symbol for the action node with which you want to associate a service call.
3. From the context menu, click Add service node. The service call is added to the activity diagram; a dotted line connects the service call to the action node. By default, the service call is synchronous.

Note: An action can only support one service call (either static or dynamic), which must be alone in the action node.

Defining a service call

After you have created the service call, you must define it. Use the Service Call Properties dialog box to specify these required properties:

- The port to which the service call is sent
- The variable that contains the business object to send
- The verb of the business object
- The correlation set (required for asynchronous inbound service calls)

Optionally, you can also specify the following:

- Support for compensation if you are using a transactional collaboration. For more information, see “Defining compensation” on page 131
- The type of service call (by default, all service calls are synchronous, but can be changed to asynchronous outbound or asynchronous inbound calls). For more information, see “Defining the service call type” on page 129.
- The time-out value to be used with synchronous and asynchronous inbound service calls. For more information, see “Defining the service call type” on page 129.
- The correlation set used for attribute matching (for synchronous and asynchronous outbound service calls).

Tip

Upon return of a service call, the business object variable contains the result of the call. The data in the original business object is lost if the service call retrieves new data for the business object. Therefore, if you anticipate needing the values in the original, it is useful to copy the original business object into a temporary variable in an action that calls the service call.

Perform the following steps to define a regular service call (one that does not use compensation or correlation sets):

1. In the activity diagram, double-click the service call symbol you have already created. The Service Call Properties dialog box is displayed, as shown in Figure 45 on page 129. Notice that you cannot directly enter a value in the Label field for a service call. Process Designer assigns the label after you complete the service call definition.

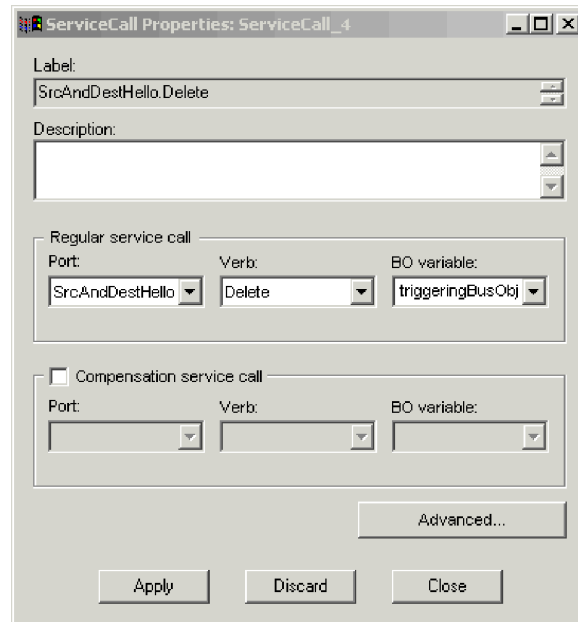


Figure 45. Service Call Properties dialog box

2. Optionally, specify the description for this service call.
3. Use the Port drop-down list to select the port the service call is going to use for sending or receiving requests.
4. Use the Verb drop-down list to specify which verb is going to be used in the request. For example, to update an application with the data contained in the business object the service call sends, use the Update verb.
5. Use the BO variable drop-down list to select the variable that contains the business object the service call sends. If you plan to support long-lived business processes, this variable must be a global template or port variable; scenario variables cannot be used in this capacity for long-lived business processes.
6. Click Apply to save the definition.

Defining the service call type

To change the type of the service call, do the following:

1. If the Service Call Properties dialog box is not open, display it by double-clicking the service call symbol in your activity diagram.
2. Ensure that you have supplied the required port name, verb, and business object variable name for the service call.
3. Click the Advanced button on the Service Call Properties dialog box. The Service Call Advanced Properties dialog box is displayed.

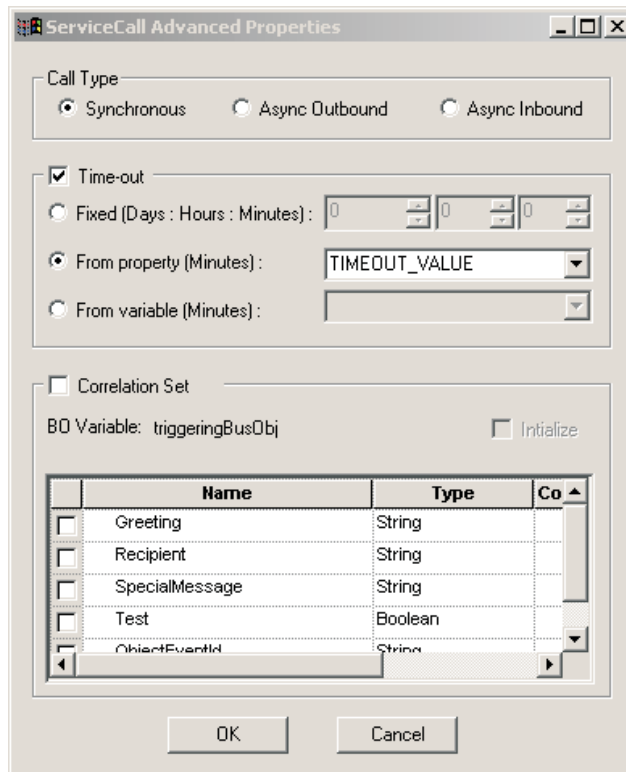


Figure 46. Service Call Advanced Properties dialog box

4. In the Call Type box, click the radio button next to the type of service call you want to use.

Note: The Async Inbound option is available only if you have enabled support for long-lived business processes in the template definition.

5. If you are using a synchronous service call or an asynchronous inbound service call and you want to specify a timeout value to be used with long-lived business processes, click the Time-out checkbox and set the timeout value to one of the following:
 - Fixed—This option requires you to specify the timeout value in days, hours, and minutes. Select this option if you want to always use the same timeout value for the service call. This value cannot be changed during collaboration configuration.
 - From property—This option enables you to use a collaboration-specific property to dynamically specify the timeout value (in seconds) during collaboration configuration. In the From Property drop-down list, select the property you have created to represent the timeout value.
 - From variable—This option enables you to use a global Java object type variable to set the timeout value (measured in seconds) during runtime. Select the appropriate variable name from the From Variable drop-down list.

Note: If the port of a synchronous service call is bound to a collaboration instead of a connector, any specified timeout value is ignored.

6. Click Ok to close the Service Call Advanced Properties dialog box.
7. Click Apply in the Service Call Properties dialog box to save your changes.

Defining compensation

Subtransaction steps of a collaboration define the transactional behavior of a transactional collaboration. A *subtransaction* is an operation in which the collaboration sends a request that causes a transactional data change in an application data store. Service calls implement subtransaction steps. A service call that has a Create, Delete, or Update request is a subtransaction step; a Retrieve request is not (because it does not change the data). Other verbs might or might not be transactional, depending on whether they change data in an application data store.

Note: Although a service call with a Retrieve request is not considered a subtransaction step, you can still specify compensation for it.

To create a collaboration template that supports transactional behavior, you define *compensation* for each subtransaction step. Compensation is a logical undo action: if execution of a collaboration object fails, it causes rollback of the previously executed operations. When rollback occurs, the collaboration runtime environment steps backward through the path of execution, executing a compensation step for every normal step that has already executed and that has compensation defined. Rollback thereby logically returns data to the state that it was in before the transactional collaboration started to execute.

Viewed in Process Designer, a subtransaction step is a service call that requests an operation such as Create, Update, or Delete. These operations always result in data-altering transactions within an application. You can also specify compensation for a service call that requests a Retrieve operation, though compensation is not required since no data is modified during a Retrieve. A service call is defined by a business object with a particular verb, which the collaboration sends either to another collaboration or to a connector. The compensation for that operation is another business object and verb. Any business object and verb can compensate for a service call.

Table 41, taken from the manual *Technical Introduction to IBM WebSphere InterChange Server*, lists some common types of compensation.

Table 41. Compensation Examples

Action	Compensation
Create a business object	Delete a business object
Delete a business object	Create a business object
Update a business object	Update a business object, restoring the former values

Compensation is supported for both synchronous service calls and asynchronous outbound service calls. If the collaboration is transactional, and if the verb for this service call requests a data modification, you can specify the compensation operation that rolls back the regular service call. Define the compensation as follows:

1. If the Service Call Properties dialog box is not open, display it by double-clicking the service call symbol in your activity diagram.
2. Click the Compensation checkbox. The Port, Verb, and BO Variable fields in the Compensation Service Call box become active.
3. In the Compensation Service Call box, select the port, verb, and business object variable to be used in the compensation service call. Compensation service calls

can use the same port, verb, and business object as the regular service call, or you can specify a different port, verb, and business object.

4. Click Apply to save your changes.

For more information on transactional collaborations, see “Using transactional features” on page 143.

Using a correlation attribute

A correlation attribute is used to identify a conversation. A *conversation* is a unit of coherent communication between two business processes. Because there can be multiple conversations when two or more business processes communicate with each other, it is necessary to identify a conversation with a correlation attribute. You can think of a correlation attribute as a UID for a conversation. The ID must be initialized when the conversation is started, and all subsequent participants must use this ID when involved in the conversation.

InterChange Server supports only one conversation per scenario. If you need more than one conversation, you must use multiple scenarios within the collaboration template, or use multiple collaboration templates. In addition, a correlation attribute can be initialized only once in the scenario.

In order to use a correlation attribute, you must ensure the following conditions are true:

- You have added support for long-lived business processes when you defined the collaboration template.
- You have created one or more template variables that are to be used for capturing correlation attribute values. For each business object attribute you plan to select for correlation, you must have a unique template variable that can capture that value.

When using a correlation attribute with a service call, the business object is determined automatically; it is always the business object assigned to the port used for the service call.

After a correlation attribute is initialized in a scenario, you can set it for outbound service calls (see “Setting correlation attributes” on page 133), match it on asynchronous inbound service calls (see “Matching correlation attributes” on page 133), or both.

Initializing correlation attributes: The first step in using correlation attributes is initialization. When you initialize a correlation attribute, you specify the business object attribute and the template variable to be used to capture that attribute’s value. Initialization must be done prior to setting the correlation attribute on an outbound service call, or matching the correlation attribute in the runtime environment.

Initialization can be done in a start node, an asynchronous outbound service call, or a synchronous service call.

Perform the following steps to define and initialize correlation attributes in a service call:

1. Ensure that Process Designer is open, and that the Service Call Properties dialog box is displayed.
2. Click Advanced to open the Service Call Advanced Properties dialog box.

3. Click Correlation Set. Notice that the business object variable is automatically defined as the one assigned to the service call's outbound port.
4. Click Initialize.
5. Click the checkbox next to each business object attribute you want to use for correlation.
6. For each selected attribute, use the drop-down list in the Correlation column to select a template variable to capture and store the attribute's value.
7. Click Apply.

The procedure for initializing a correlation set from the start node is the same, with the exception of clicking the Initialize checkbox. Since a start node does not participate in matching correlation attributes, initialization is implicit.

Setting correlation attributes: After you have defined and initialized a correlation set, you can assign it to any outbound service call. The service call then sends out a request that includes the correlation set. You can set correlation attributes on as many outbound service calls as you want.

Perform the following steps to set correlation attributes on an outbound service call:

1. Ensure that Process Designer is open, and that the Service Call Properties dialog box is displayed for the outbound service call.
2. Click Advanced to open the Service Call Advanced Properties dialog box.
3. Click Correlation Set.
4. Click the checkbox next to each business object attribute you defined in your initialized correlation set.
5. For each selected attribute, use the drop-down list in the Correlation column to select the template variable used to capture and store the attribute's value. This variable must be the same one used when the correlation set was initialized.
6. Click Apply.

Matching correlation attributes: After you have initialized a correlation set, you can configure an asynchronous inbound service call to receive any response that matches that particular correlation set.

Perform the following steps to specify a correlation set in an asynchronous inbound service call:

1. Ensure that Process Designer is open, and that the Service Call Properties dialog box is displayed for the asynchronous inbound service call.
2. Click Advanced to open the Service Call Advanced Properties dialog box.
3. Click Correlation Set.
4. Click the checkbox next to each business object attribute you defined in your initialized correlation set.
5. For each selected attribute, use the drop-down list in the Correlation column to select the template variable used to capture and store the attribute's value. This variable must be the same one used when the correlation set was initialized.
6. Click Apply.

At runtime, if the asynchronous inbound service call has attributes that match those defined in the correlation set, the service call is invoked by the scenario. If its attributes do not match, the service call is routed to another scenario or collaboration that has defined a matching correlation set.

Handling results

When a service call executes, the scenario receives two return values: a status and a business object. Table 42 describes the use of each.

Table 42. Service call return values

Returned	Description
Status	The normal outgoing transition link of the action that generated the service call tests for success of the service call. An exception transition link can check for a <code>ServiceCallException</code> exception to test for failure of the service call. A service call can fail due to transport problems as well as application problems. Because transport failure can cause duplication of data, it is important to determine whether a service call failure was due to transmission problems. For more information, see “Handling particular service-call exceptions” on page 178.
Business object	Upon completion of a service call, the business object variable that the service call used contains new data values if the service call resulted in any application change.

- After a Create service call, you need not check the value of the business object. If the service call returns successfully, the Create operation succeeded.
- A Delete request does not necessarily cause the actual deletion of application data. Because many applications do not support deletion, a connector handles Delete verbs according to the rules of its application. For example, a connector might translate a Delete request into an Update request, updating the application entity to inactive status.

Performance considerations

Collaboration performance is affected by the number of service calls and the size of the business objects passed by the service calls. Although you cannot change business object size, try to reduce the number of service calls that a collaboration makes.

For example, suppose the scenario needs to perform an operation on child business objects in a hierarchical business object. In some situations, it can be more efficient to retrieve the entire hierarchical object and iterate through it locally, rather than creating a service call for each child business object that needs to perform a service call.

Subdiagrams

When logic of the activity diagram gets complex, it is often useful to partition the logic, separating discrete units of logic into *subdiagrams*. Each subdiagram is associated with a particular main diagram.

Figure 47 illustrates a scenario in which the main activity diagram contains references to two subdiagrams, Retrieve Subdiagram and Delete Subdiagram.

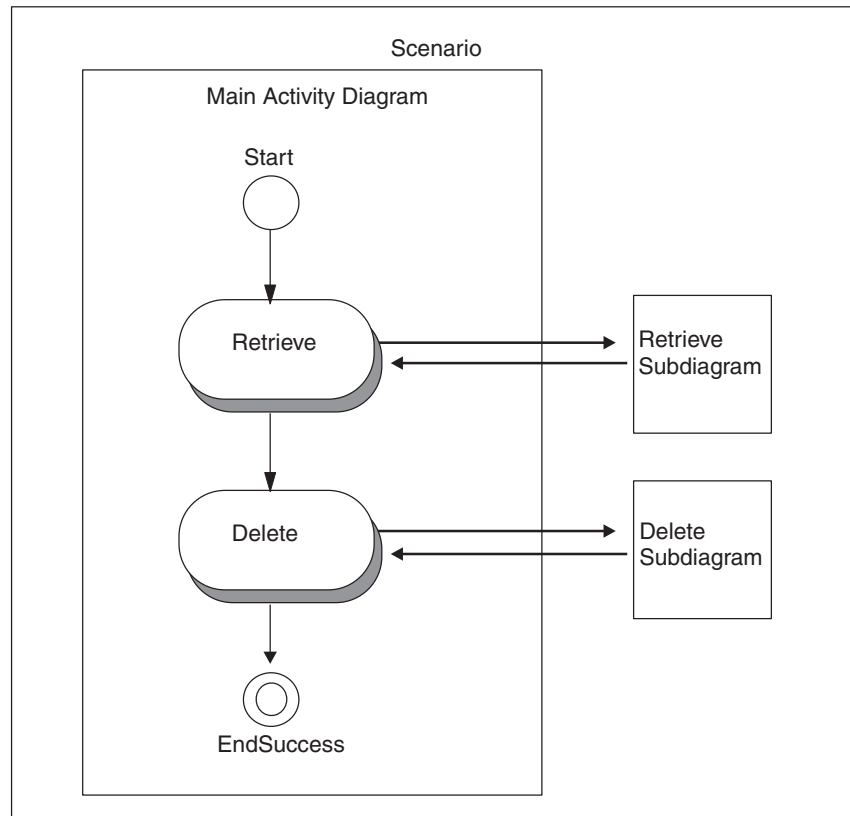


Figure 47. Scenario with two subdiagrams

Note: An iterator is a specialized form of subdiagram. All basic information about subdiagrams applies to iterators; for information that applies specifically to iterators, refer to “Iterators” on page 139.

The activity diagrams in a scenario are hierarchically arranged. All subdiagrams and iterators in a scenario descend from the scenario’s main activity diagram. Figure 48 illustrates this relationship. The activity diagram in which the subdiagram symbol appears is referred to as the *parent diagram* for the subdiagram.

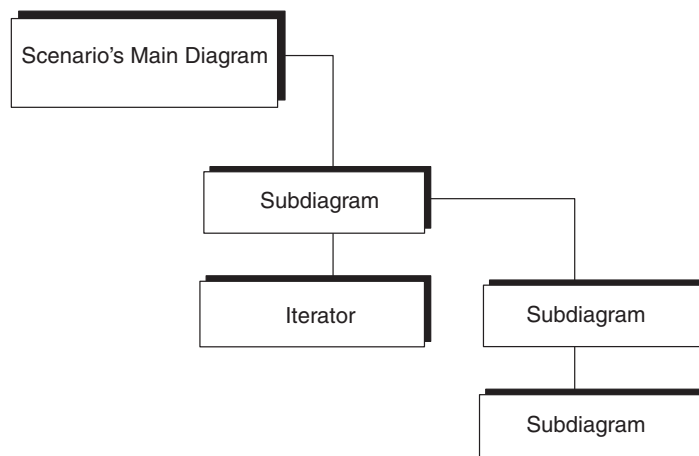


Figure 48. Relationship of main diagram and subdiagrams

A subdiagram has access to all collaboration template properties and to all scenario variables. Table 43 summarizes how a subdiagram differs from the main diagram.

Table 43. Comparison of main diagram and subdiagrams

Issue	Main diagram	Subdiagram
How it is created at design time	Automatically created when you create a scenario	Controlled by the subdiagram symbol in the parent diagram
Cause of execution at runtime	Starts execution when the collaboration runtime environment hands it a triggering event	Starts executing when the parent diagram's execution path leads to it; has no triggering event
What happens to unhandled or raised exception	Passes to the collaboration runtime environment	Passes to the parent diagram
Completion at runtime	Returns to the collaboration runtime environment	Returns to the parent diagram

Creating a subdiagram

To add a subdiagram to the activity diagram:

1. In the Symbols toolbar, click the Subdiagram button.
2. Click in an active activity diagram to place the subdiagram symbol.

A unique identifier appears for the subdiagram in the scenario tree, hierarchically arranged under the parent diagram. The scenario tree displays the name in the following format:

(UID)

If you provide a label for the subdiagram, the scenario tree displays the name in the following format:

label (UID)

The UID is a unique identifier that is also the name of the subdiagram object in the scenario tree. As with UIDs for other symbols, you can choose whether or not to display the UID for the subdiagram. To turn on or off display of the UID, use the context menu on the scenarios node in the template tree.

3. Double-click the subdiagram name in the scenario tree, or right-click its node in the diagram editor window and select Open Subdiagram.

A new window displays in the Working Area in which to define a new activity diagram.

Like the main activity diagram, the subdiagram starts with a Start symbol and ends with an End Success symbol, and, optionally, one or more End Failure symbols. A subdiagram can contain all diagramming components, including subdiagrams and iterators.

Defining a subdiagram

After the subdiagram appears in the activity diagram, you can define its properties in the Subdiagram Properties dialog. The properties of a subdiagram are its label and description. All are optional.

To define subdiagram properties:

1. Display the Subdiagram Properties dialog.
You can display this dialog in any of the following ways:
 - Double-click on the selected subdiagram.
 - Right-click to bring up the context menu, then choose Properties.
 - Choose Properties from the Edit pull-down menu.

- Use the keyboard shortcut Ctrl + Enter.
2. Optionally, specify the label and description for this subdiagram.
The label makes the activity diagram more readable, by labeling the subdiagram with text that is more descriptive than the UID. The description field is a place for a comment, which describes the purpose of the subdiagram.
 3. Click Apply to save the subdiagram properties. Click Discard to clear the properties. Click Close to cancel this subdiagram definition.

Deleting a subdiagram

To delete a subdiagram, display its parent diagram or subdiagram and do the following:

1. Select the symbol of the subdiagram to delete.
2. Choose the Delete option from the Edit menu. Alternatively, you can click the DEL (Delete) key.

If the parent diagram is expanded in the scenario tree, the subdiagram name disappears from the scenario tree when it is deleted.

Handling subdiagram completion status

The execution of a parent diagram responds to the execution status of its subdiagrams. It is the subdiagram's developer who decides the subdiagram's completion status, as well as its exception-handling behavior. A collaboration can intentionally end a subdiagram in either of the following ways:

- Place the End Success node at the end of the subdiagram's execution path to indicate successful subdiagram execution.
- Place an End Failure node at the end of the subdiagram's execution path to indicate unsuccessful subdiagram execution.

Handling successful subdiagram execution

The End Success termination node indicates that execution has completed successfully. When a subdiagram ends with an End Success node, the collaboration runtime environment ends the subdiagram and passes control to the parent diagram. The flow of the parent diagram proceeds to the next node after the subdiagram node. This next node is usually a decision node that tests the status of the subdiagram. This decision node can include any of the following branches to test if the subdiagram execution:

- A normal branch that tests for boolean conditions that you can set.
If the collaboration's execution is in the Normal state, the collaboration runtime environment evaluates the condition of the normal branch. The default branch executes if the normal branch does not evaluate to true.
- Exception branches test for specific raised exceptions.
If the collaboration's execution is in the Exception state, the collaboration runtime environment evaluates the conditions of the exception branches.

A subdiagram can end successfully in the following ways:

- The subdiagram completes its task and does *not* encounter any exceptions.
The collaboration's execution is in the Normal state. When control passes to the parent diagram and the parent diagram's next node is a decision node, the collaboration runtime environment evaluates the normal branch.
- The subdiagram encounters an exception, handles it (without raising the exception), and intentionally ends in success.

The collaboration's execution is in the Normal state. When control passes to the parent diagram and the parent diagram's next node is a decision node, the collaboration runtime environment evaluates the normal branch.

- The subdiagram encounters an exception, handles it by raising the exception to the parent diagram, and intentionally ends in success.

In this case, the collaboration's execution is in the Exception state. When control passes to the parent diagram and the parent diagram's next node is a decision node, the collaboration runtime environment evaluates any exception branches. The exception branch should lead to an action node that handles the exception. The best way to handle the exception is to raise it again and include the exception text as the reason for the exception.

In a collaboration with multiple levels of activity diagrams, you must explicitly use the `raiseException()` method to raise an exception up through each level in the activity diagram to provide the original exception text in the list of unresolved flows. Each subsequent `raiseException()` call raises the exception and passes the original exception text. When execution reaches the main diagram, the collaboration runtime environment performs its exception-handling operation, such as writing to the log or, if the collaboration is transactional, initiating rollback.

Note: The developer can have a subdiagram end in success even if it encounters an exception, as long as it handles the exception. Successful completion means only that execution of the subdiagram reached an End Success symbol and that code was available to handle any exceptions that were raised.

For information on how to create a decision node, see "Decision nodes" on page 120. For more information on how to implement exception handling, see Chapter 7, "Handling exceptions," on page 167.

Handling unsuccessful subdiagram execution

The End Failure termination node indicates that execution has *not* completed successfully. When a subdiagram ends with an End Failure node, the collaboration runtime environment ends the subdiagram as well as the entire collaboration. Control passes to the collaboration runtime environment, which makes an entry in the collaboration's log destination and creates an unresolved flow.

A subdiagram can end unsuccessfully in the following ways:

- The subdiagram encounters an exception, handles it, and intentionally ends in failure. In this case, the subdiagram should include the exception text as the reason for the exception.
- The subdiagram encounters an unexpected exception that it does *not* handle, and intentionally ends in failure.

Whenever a subdiagram ends with the End Failure node, the collaboration runtime environment terminates the *entire* collaboration. For information on how to handle exceptions encountered in a subdiagram, see "Successfully ending a subdiagram or iterator" on page 171. For information on how the collaboration runtime environment creates unresolved flows, see "Successfully ending the main diagram" on page 170.

Iterators

An *iterator* is a specialized form of subdiagram that implements loops or iterations. Use an iterator to perform an operation:

- On all the attributes of a business object
- On all the elements of a business object array

An iterator can also be used as a loop. Any values required for initializing, testing, and incrementing the loop must be supplied in the Iterator Properties dialog box.

An iterator diagram can call subdiagrams or other iterators. A hierarchy of iterators is necessary to process hierarchical business objects or hierarchical business object arrays.

When execution of the parent diagram reaches the Iterator symbol, control passes to the iterator activity diagram. The collaboration repeats execution of the iterator diagram for each attribute in the business object or each business object in the business object array. You have access to the item currently in the iterator through the iterator variable specified in the Iterator Properties dialog.

After the iterator has finished executing, control passes back to the parent diagram.

Creating an iterator

To add an iterator to the activity diagram:

1. In the Symbols toolbar, click the Iterator button.
2. Click in the workspace to place the Iterator symbol.

A unique identifier appears for the iterator in the scenario tree, hierarchically arranged under the parent diagram. The scenario tree displays the name in the following format:

(UID)

If you provide a label for the iterator, the scenario tree displays the name in the following format:

label (UID)

The UID is a unique identifier that is also the name of the iterator object in the scenario tree. As with UIDs for other symbols, you can choose whether or not to display the UID for the iterator. To turn on or off display of the UID, use the context menu on the scenarios node in the template tree.

Creating iterator variables

An iterator working on attributes of a business object or an array of business objects needs to have a variable to hold the item being processed in each iteration. This iterator variable is actually a scenario variable that is created and initialized in the Scenario Definitions dialog box. Create iterator variables before defining the properties of an iterator.

If the iterator is working on the attributes of a business object, you can use an Object to hold the current attribute. For example, you might have the following declaration:

```
Object iterAttr = null;
```

If the iterator is working on business objects in an array, you can use a variable of type BusObj to hold the current business object. For example:

```
BusObj iterBusObj = new BusObj("LineItem");
```

If the iterator is being used as a loop, you do not need to create an iterator variable. The system creates one automatically during processing. The loop index variable can then be retrieved with the `getCurrentLoopIndex()` API.

Defining an iterator

After the iterator appears in the activity diagram, you can define its properties in the Iterator Properties dialog box (see Figure 49).

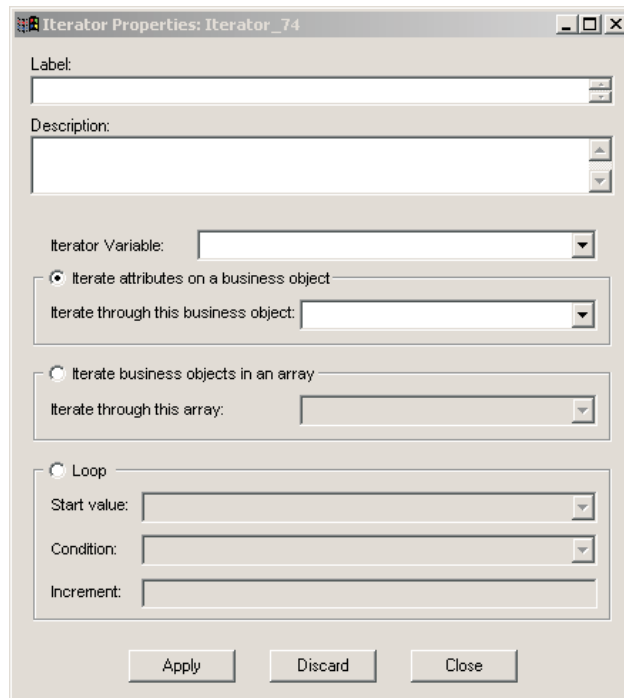


Figure 49. Iterator Properties dialog box

Open the Iterator Properties dialog box by right-clicking the iterator and selecting Properties from its context menu. You can define the iterator's label and provide a description; both of these properties are optional. However, you must define certain other properties depending on the type of iterator you want to use. The following sections describe the requirements for defining each type of iterator.

Using an iterator on attributes of a business object

If you want to iterate attributes of a business object, do the following:

1. Use the Iterator Variable field to specify the variable that is going to hold the item being processed during an iteration. The drop-down list in this field contains all template and scenario variables. You can select the variable from the list, or type the variable name directly in the field.
2. Click the Iterate attributes in a business object radio button.
3. Use the Iterate through this business object field to specify the business object whose attributes you want to iterate through. Either type the name directly in the field or use the drop-down list to select the business object.
4. Click Apply.

After you have defined an iterator's properties, you must edit its activity diagram to define what the iterator does for each attribute it processes. Open an iterator's activity diagram by double-clicking the iterator name in the template tree view of Process Designer.

Using an iterator on business objects in an array

If you want to iterate business objects in an array, do the following:

1. Use the Iterator Variable field to specify the variable that is going to hold the item being processed during an iteration. The drop-down list in this field contains all template and scenario variables. You can select the variable from the list, or type the variable name directly in the field.
2. Click the Iterate business objects in an array radio button.
3. Use the Iterate through this array field to specify the array you want to iterate through. Either type the name directly in the field or use the drop-down list to select the array.

Use the following syntax if you type the value directly in the field:

BusinessObjectVariable.AttributeName

where *BusinessObjectVariable* is the name of the parent business object and *AttributeName* is the name of the attribute that represents the array of child business objects.

For example, to iterate through business objects in an array represented by the Items attributes in a business object contained in the variable order, type order.Items in the Iterate through this array field.

4. Click Apply.

After you have defined an iterator's properties, you must edit its activity diagram to define what the iterator does for each business object it processes. Open an iterator's activity diagram by double-clicking the iterator name in the template tree view of Process Designer.

Using an iterator as a loop

To define a loop, do the following:

1. Click the Loop radio button.
2. Use the Start Value field to specify the initial value of the counter variable. You can use the drop-down list to select a variable that holds the value, or you can type the value directly in the field.
3. In the Condition field, specify the condition that must be true in order for the loop to execute. The drop-down list contains all boolean variables defined in the template. Select the variable you want to use, then type the condition directly into the field.
4. Use the Increment field to specify the method of incrementing the value of the counter variable.
5. Click Apply.

After you have defined an iterator's properties, you must edit its activity diagram to define what the iterator does for each attribute or business object it processes. Open an iterator's activity diagram by double-clicking the iterator name in the template tree view of Process Designer.

Adding a break

A break can be added to an iterator's activity diagram to force premature termination of the iteration. When the iterator's execution path reaches the break

symbol, the iterator terminates and control is passed back to the parent diagram. Breaks can be used with all types of iterators.

Place a break symbol in an iterator's activity diagram as follows:

1. Click the Break button in the Symbols toolbar.
2. In the activity diagram, position your cursor where you want to place the break, and then click. The break symbol is added to the diagram.



Figure 50. Break symbol

Optionally, you can add a label and description to a break symbol. Double-click the break symbol in an activity diagram to open its Break Properties dialog box and edit the properties as desired.

Using other features of the Symbols toolbar

The Diagram Symbols toolbar contains a Select button and a Text button. Use the Text button to add a text box to your activity diagram, as described in “Using the text box feature.” Use the Select button to cancel an operation, as described in “Cancelling an operation.”

Using the text box feature

You can insert a text box in an activity diagram and place text in the box, or edit existing text. To add text to an activity diagram:

1. In the Symbols toolbar, click the Text button.
2. Click in the workspace to place the Text symbol.
A text box appears with the word “Text” within the box.
3. Double-click the word “Text” and enter any text you like.

You can select and drag the text box to position it at a different point within the diagram.

Cancelling an operation

If you change your mind about inserting a particular symbol in an activity diagram, you can cancel the insertion. To cancel an operation after you have clicked its button in the Diagram Symbols toolbar, click the Select button in the Symbols toolbar.

To cancel a transition link, use the Esc key. Each press of the Esc key undoes the last segment of the transition link.

Obtaining values of collaboration configuration properties

Action nodes and decision nodes commonly need to obtain and evaluate the values that an implementer or developer configures for a collaboration object's configuration properties. The collaboration object inherits its configuration properties from the collaboration template. For more information on how to define collaboration properties, see “Defining collaboration configuration properties (the Properties tab)” on page 91.

To obtain the value of a property, a scenario calls the `getConfigProperty()` method. If the property is a list of values, a scenario calls `getConfigPropertyArray()`. For more information, see “Retrieving a collaboration configuration property” on page 197.

Using transactional features

A **transactional collaboration** rolls back if it encounters an error that stops its execution. For rollback to be successful, each scenario must have compensation specified for each subtransaction step. To set the transactional properties of a collaboration, you must take the steps outlined in Table 44.

Table 44. Defining a transactional collaboration

Definition step	Description	For more information
Assign a minimum transaction level for the collaboration template.	The transactional collaboration uses the minimum transaction level to determine when to perform transaction rollback.	“Specifying the minimum transaction level” on page 84
Specify compensations for subtransaction steps.	The transactional collaboration uses the compensation defined for its subtransactions to perform the actual rollback.	“Defining compensation” on page 131

For a more detailed explanation of transactional collaborations, refer to the chapter on transactional collaborations in the *Technical Introduction to IBM WebSphere InterChange Server*.

Terminating the execution path

Each execution path in an activity diagram must terminate in either success or failure.

Terminating in success

Ending the collaboration in success means that the execution path for the activity diagram successfully handled the triggering event. Either all the execution was successful or, if an exception occurred, the activity diagram handled it in a way that still allowed the collaboration to end in success. For more information, see Chapter 7, “Handling exceptions,” on page 167.

To indicate successful termination, place the End Success symbol at the end of the execution path. When the collaboration runtime environment executes the End Success, it terminates the current execution path and passes control to the next higher level of execution (the parent diagram), if one exists. When a subdiagram or iterator reaches an End Success node, control passes to the parent diagram. When the main activity diagram reaches an End Success node, control passes to the collaboration runtime environment, which performs its own error-handling actions.

Adding the End Success symbol

To terminate an execution path in success, place an End Success symbol in the activity diagram and connect it. To add an End Success symbol to an activity diagram:

1. In the Diagram Symbols toolbar, click the End Success button.
2. Click in the workspace to place the End Success symbol.

Defining the End Success symbol

After the End Success symbol appears in the activity diagram, you can define its properties in the End Success Properties dialog. The properties of an End Success symbol are its label and description. Both are optional.

To define End Success properties:

1. Display the End Success Properties dialog.

You can display this dialog in any of the following ways:

- Double-click on the selected End Success node.
- Right-click to bring up the context menu, then choose Properties.
- Choose Properties from the Edit pull-down menu.
- Use the keyboard shortcut `Ctrl + Enter`.

The End Success Properties dialog displays.

2. Optionally, specify the label and description for this action node.

The label makes the activity diagram more readable, by labeling the End Success symbol with text that is more descriptive than the UID. The description field is a place for a comment, which describes the purpose of the End Success symbol.

3. Click Apply to save the End Success properties. Click Discard to clear the properties. Click Close to cancel this End Success definition.

Terminating in failure

Ending the collaboration in failure means that the activity diagram was unable to execute properly and execution must stop. To indicate unsuccessful termination, place the End Failure symbol at the end of the execution path. When the collaboration runtime environment executes the End Failure, it terminates the entire collaboration. When a subdiagram or iterator reaches an End Failure node, both the subdiagram or iterator terminates as well as all parent diagrams terminate. The collaboration runtime environment then performs its own error-handling actions.

Note: Reaching an End Failure symbol *always* stops execution of the collaboration. However, ending in End Failure does not automatically fail the triggering event. Only if the collaboration's execution is in the Exception state does the collaboration runtime environment create an unresolved flow for the triggering event. For more information, see Chapter 7, "Handling exceptions," on page 167.

Adding the End Failure symbol

To terminate an execution path in failure, place an End Failure symbol in the activity diagram and connect it. To add an End Failure symbol to an activity diagram:

1. In the Diagram Symbols toolbar, click the End Failure button.
2. Click in the workspace to place the End Failure symbol.

Defining the End Failure symbol

After the End Failure symbol appears in the activity diagram, you can define its properties in the End Failure Properties dialog. The properties of an End Failure symbol are its label and description. Both are optional.

To define End Failure properties:

1. Display the End Failure Properties dialog.

You can display this dialog in any of the following ways:

- Double-click on the selected End Failure node.
 - Right-click to bring up the context menu, then choose Properties.
 - Choose Properties from the Edit pull-down menu.
 - Use the keyboard shortcut Ctrl + Enter.
2. Optionally, specify the label and description for this action node.

The label makes the activity diagram more readable, by labeling the End Failure symbol with text that is more descriptive than the UID. The description field is a place for a comment, which describes the purpose of the End Failure symbol.
 3. Click Apply to save the End Failure properties. Click Discard to clear the properties. Click Close to cancel this End Failure definition.

Other activity diagram operations

This section provides information on the following additional operations that Process Designer provides on an activity diagram:

- “Opening and closing activity diagrams”
- “Documenting an activity Diagram”
- “Copying an activity diagram” on page 146
- “Deleting within an activity diagram” on page 146

Opening and closing activity diagrams

Process Designer provides the ability to open and close an activity diagram.

Opening an activity diagram

To open an activity diagram, you can perform any of the following actions:

- Choose Open All Diagrams from the Template menu.

Open All Diagrams opens a window for each activity diagram defined for the template.
- Select a scenario, subdiagram, or iterator in the scenario tree:
 - From the Template menu, choose Open Diagram to open a window for the associated activity diagram.
 - Double-click on the scenario, subdiagram, or iterator name.
- If the activity diagram is a subdiagram or an iterator, you can open its parent diagram by right-clicking in the workspace to display the context menu and choosing Open Parent Diagram.

Closing an activity diagram

To close an activity diagram, you can perform any of the following actions:

- Choose Close All Diagrams from the Template menu.

When at least one activity diagram is open, Close All Diagrams closes all windows for activity diagrams.
- Select the close button at the top of the window that displays the scenario, subdiagram, or iterator.

Documenting an activity Diagram

There are two ways to document an activity diagram:

- Print a graphic representation of a diagram, subdiagram, or iterator.
- Save the full activity diagram to a text file.

Printing a Diagram

You can print a graphic representation of each diagram, subdiagram, and iterator. Long representations print on multiple pages. To print, do the following:

1. Open the diagram—select it in the template tree view and either right-click on its name and click Open Diagram from the context menu or select Open Diagram from the Template menu.
2. Select Print from the File menu or use the shortcut key combination Ctrl+P.

Saving a diagram as a text file

You can save the entire activity diagram as a text file. To do so:

1. Choose Save Diagram As Text File... from the Template menu.
2. In the Save Diagram As Text dialog box, specify the file name.

Copying an activity diagram

You cannot copy an entire activity diagram apart from its scenario, but you can copy activity-diagram contents. To copy the contents of an activity diagram:

1. Display the source activity diagram.
2. Select the objects that you want to copy.
 - To select some of the objects, press and hold the left mouse button while moving the mouse to define the rectangular area that you want to copy. Release the mouse button when you are done.
 - To select specific objects one at a time, click one object and then, holding down the Shift key, click the others. With the Shift key pressed, click an object to remove it from the selection.
 - To select all the objects, on the Edit menu, choose Select All (Ctrl+A shortcut).
3. On the Edit menu, choose Copy (Ctrl+C shortcut).
4. Display the destination activity diagram.
5. On the Edit menu, choose Paste (Ctrl+V shortcut).
6. Revise the definitions of the symbols, as necessary, so that references to properties, ports, and variables are correct.

Note that copying is not an iterative process. If you copy and paste a subdiagram or iterator, Process Designer creates an empty activity diagram in the scenario tree, subordinate to the current diagram, to represent the subdiagram or iterator diagram. You must manually copy and paste the contents of the subdiagrams and iterator diagrams. If a break symbol is selected in an iterator node during the copy operation, it can be pasted only in another iterator node.

Important: You cannot paste a duplicate Start symbol into an activity diagram. When selecting symbols to copy and paste, avoid copying the Start symbol to a diagram that already has one. If you keep the default User Preferences setting for diagrams, Process Designer automatically places the Start symbol in each new diagram. For more information, see “Changing display: user preferences” on page 187.

Deleting within an activity diagram

To delete a symbol within an activity diagram, select the symbol to delete and take one of the following actions:

- From the Edit menu, choose Delete.
- Press the DEL (Delete) key.

To delete an entire activity diagram, delete the scenario in which the activity diagram is defined. For more information, see “Deleting a scenario” on page 100. For a scenario with subdiagrams or iterators, you can delete the subdiagram or iterator by deleting its symbol from its parent diagram.

Chapter 6. Using Activity Editor

Activity Editor is a graphical interface that enables you to specify a collaboration's business logic without writing Java code. This chapter describes the Activity Editor interface, discusses the components in activity definitions, lists the supported function blocks used to create business logic and provides an example of using Activity Editor. It contains the following sections:

- "Starting Activity Editor"
- "The Activity Editor interface"
- "Activity definitions" on page 154
- "Supported function blocks" on page 157
- "Example: Changing a date format" on page 159
- "Importing Java packages and other custom code" on page 162

Starting Activity Editor

Launch Activity Editor from within an action node in a collaboration template scenario, as follows:

1. Select the action node to which you want to add business logic.
2. Open the action node's Action Properties dialog box by doing one of the following:
 - Double-clicking the action node
 - Right-clicking the action node, and then clicking Properties from the context menu that pops up
3. Click Edit. Activity Editor opens in a new window.

The Activity Editor interface

The following sections describe the views, menus, toolbars, and keyboard shortcuts that are part of the Activity Editor interface.

Activity Editor views

Activity Editor has two views for adding code fragments: Graphical and Java. The following sections describe each view.

Graphical view of Activity Editor

The Graphical view of the Activity Editor enables you to specify the action node's logic without having to write much, if any, Java code. Instead, you can drag and drop function blocks onto the workspace to represent the flow of the activity definition. Figure 51 on page 150 illustrates the Graphical view.

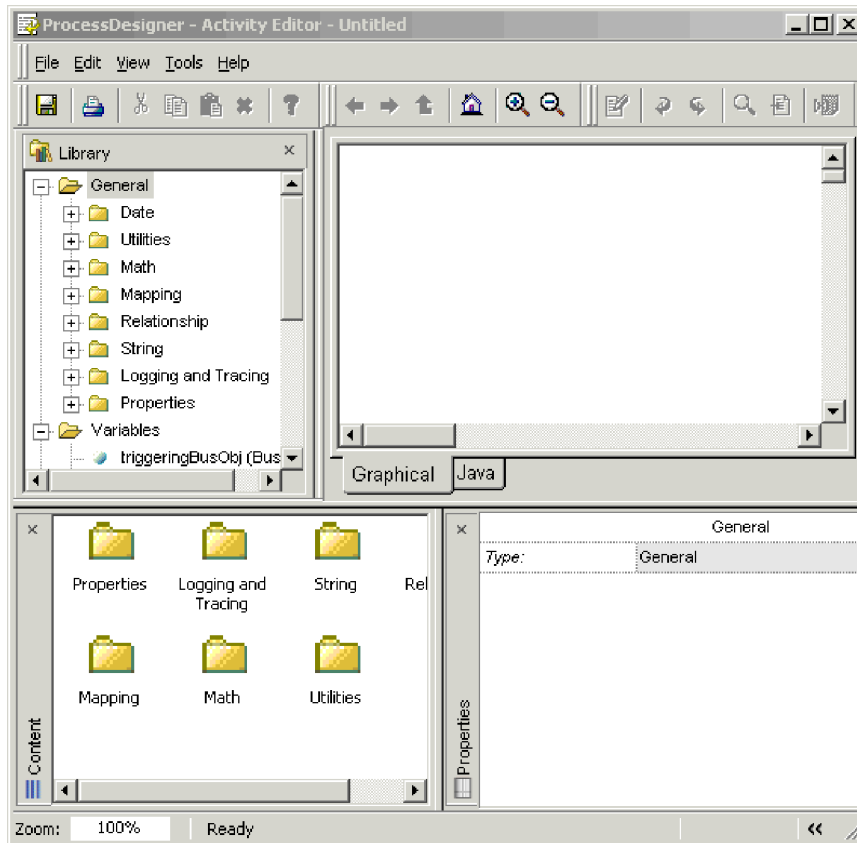


Figure 51. Activity Editor: Graphical view

The Graphical view of Activity Editor has four main windows: the Activity Workbook window, the Library window, the Content window, and the Properties window.

- Activity Workbook window—This window, usually referred to as the editing canvas, is where you drag and drop the function blocks that make up the business logic for your action node.
- Library window—This window contains a tree view of the available function blocks and, optionally, the named groups. The function blocks are arranged in folders according to their purpose (see “Supported function blocks” on page 157 for more information). In addition, this window contains the following folders:
 - System—This folder contains system elements that can be added to the editing canvas. System elements include comments, descriptions, labels, to-do tags, and constants. (See “New Constant function block” on page 156 and “Tags for activity definitions” on page 156 for more information on using these components.)
 - My Library—This folder enables you to customize the Library window.
 - My Collection—This folder enables you to create a collection of the components you use most often. You can place regular function blocks in this folder, or create your own reusable component group (see “Component groups” on page 156 for more information).
 - Variables—This folder contains global variables that are available for use in the current activity. It typically contains the port’s business object variables, as well as all of the other business objects and variables defined in the scenario.
- Content window—This window contains a large icon list of the function blocks available within the currently selected folder in the Library window. You can

select a function block to view its description and properties in the Properties window, or you can drag and drop a function block icon onto the editing canvas to create part of the activity flow.

- Properties window—This window displays the properties of the currently selected function block. Properties are displayed in a grid within this window. Some properties can be edited directly in the Properties window, while others are read-only.

Java view of Activity Editor

The Java view can be used to add traditional Java code to an action node. Use this view if you prefer to write code instead of representing it graphically, or if the Graphical view does not provide all of the functions you need for the code fragment.

Important

Once you enter or modify code in the Java view, you cannot use the Graphical view. All further work on the code fragment must be done in Activity Editor's Java view, or directly in the Code Fragment window of the Action Properties dialog box.

To simplify the reading and understanding of program code, the Java view displays code using a color-coded scheme, as described in Table 45.

Table 45. Code Fragment Color Scheme

Color	Code Type
Green	Comment
Pink	String literal
Blue	Keyword

Figure 52 illustrates the Activity Editor's Java view.

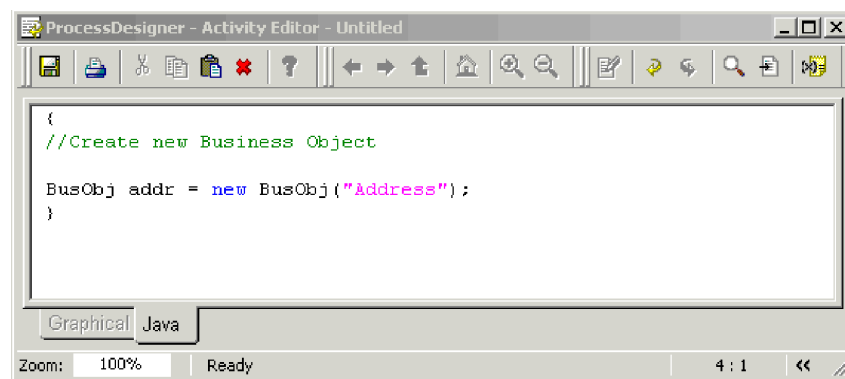


Figure 52. Activity Editor: Java view

Activity Editor menus

This section describes the functionality available from the Activity Editor menus.

File menu

The Activity Editor File menu provides the following options:

- Save—Saves the activity to Process Designer Express.

- Print Setup—Opens the Print Setup dialog box for specifying print options.
- Print Preview—Switches the editor to print preview mode. This menu option is available only in the Java view.
- Print—Opens the Print dialog box for printing the current activity. This menu option is available only in the Java view.
- Close—Closes the Activity Editor.

Edit menu

The Activity Editor Edit menu provides the following options:

- Undo—Clears the last change you made and restores the previous version. This menu option is available only in the Java view.
- Redo—Restores a change that was previously removed with the **Undo** command. This menu option is available only in the Java view.
- Cut—Deletes the selected item. The item is copied to the clipboard.
- Copy—Copies the selected item to the clipboard.
- Paste—Pastes the object currently in the clipboard into the activity at the point where the cursor is located.
- Delete—Deletes the selected object.
- Select All—Selects all objects in the activity.
- Find—Finds specified text in the editing area. This menu option is available only in the Java view.
- Replace—Finds specified text and replaces it with the new text you specify. This menu option is available only in the Java view.
- Goto Line—Moves the cursor to a specified line. This menu option is available only in the Java view.

View menu

The Activity Editor View menu provides the following options:

- Design mode—Toggles between Design mode and Quick View mode.
- Quick View mode—Toggles between Quick View mode and Design mode.
- Go To—Opens the following submenu for navigating within the activity:
 - Back—Moves backward in the navigation history.
 - Forward—Moves forward in the navigation history.
 - Up One Level—Displays the diagram from one higher level.
 - Home—Goes to the top-level diagram in the Graphical view.
- Zoom In—Magnifies content in the editor.
- Zoom Out—Minimizes content in the editor.
- Zoom To—Opens the Zoom dialog box for specifying a particular level of zoom.
- Library window—Toggles the Library window on and off.
- Content window—Toggles the Content window on and off.
- Properties window—Toggles the Properties window on and off.
- Toolbars—Opens the following submenu for displaying and closing toolbars:
 - Standard—Toggles the Standard toolbar on and off.
 - Graphics—Toggles the Graphics toolbar on and off.
- Status Bar—Toggles the status bar on and off.
- Preferences—Opens the Preferences dialog box for specifying the Activity Editor's default behavior.

Tools menu

The Activity Editor Tools menu provides the following option:

- Translate—Translates the current activity to Java code and opens the Java view.

Help menu

The Activity Editor Help menu provides the following options:

- Help Topics—Opens the help topics.
- Documentation—Opens the InterChange Server documentation.

Context menu

The Activity Editor Context menu is accessed by right-clicking on the editing canvas. It provides the following options:

- New Constant—Creates a new Constant component on the editing canvas.
- Add Label—Creates a new label component on the editing canvas.
- Add Description—Creates a new description component on the editing canvas.
- Add Comment—Creates a new comment component on the canvas.
- Add To Do—Creates a new reminder component in the activity.
- Add To My Collection—Creates a new component group for reuse in the Library window.

Activity Editor toolbars

Activity Editor has two toolbars: the Standard toolbar and the Graphics toolbar.

The Standard toolbar provides functionality for saving and printing activities, cutting, copying, pasting, and deleting elements in the activity, and accessing help.

The Graphics toolbar provides functionality for navigating within activities. The buttons correspond to the View → Zoom and View → Go To menu items.

Activity Editor keyboard shortcuts

Table 46 lists the Activity Editor menu items and the keyboard shortcuts associated with them.

Table 46. Activity Editor keyboard shortcuts

Menu	Menu item	Keyboard shortcut
File menu	Save	Ctrl+S
	Print Setup	Ctrl+Shift+P
	Print Preview	No shortcut available
	Print	Ctrl+P
	Close	No shortcut available
Edit	Cut	Ctrl+X
	Copy	Ctrl+C
	Paste	Ctrl+P
	Delete	Del
	Select All	Ctrl+A
	Find	Ctrl+F
	Goto Line	Ctrl+G

Table 46. Activity Editor keyboard shortcuts (continued)

Menu	Menu item	Keyboard shortcut
View	Design Mode	No shortcut available
	Quick View Mode	No shortcut available
	Go To/Back	Alt+Left Arrow
	Go To/Forward	Alt+Right Arrow
	Go To/Up One Level	No shortcut available
	Go To/Home	Alt+Home
	Zoom In	Ctrl++
	Zoom Out	Ctrl+ -
	Zoom To	Ctrl+M
	Library Window	No shortcut available
	Content Window	No shortcut available
	Properties Window	No shortcut available
	Toolbars	No shortcut available
	Status Bar	No shortcut available
Preferences	Ctrl+U	
Tools	Translate	Ctrl+T
Help	Help Topics	F1
	Documentation	No shortcut available

Activity definitions

Activity Editor is used to create *activity definitions*, which specify the business logic for each action node in the collaboration template. Each action node has one activity definition associated with it.

Function blocks

An activity definition is based on *function blocks*. A function block represents a discrete part of an activity definition, such as a constant, a variable, or a particular piece of functionality (like a programming method). Many of the function blocks in Activity Editor correspond to individual methods in the Collaboration API.

Function blocks are placed on the editing canvas by dragging and dropping them from either the Library or Content window. Once a function block is dropped on the editing canvas, you can move it around as needed. Just click the function block icon on the canvas to select it and drag it to the desired location.

Function blocks can have inputs, outputs, or both. The inputs and outputs for each function block are predefined, and accept only the specified value type. When the function block is dropped on the editing canvas, its input and output ports are represented by arrows, as shown in Figure 53 on page 155.

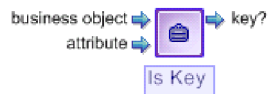


Figure 53. A function block with input and output ports

These ports serve as connecting points for linking between the function block and other components. By default, the name of each input and output is displayed next to its connection port (you can use the View → Preferences option to hide the names).

Web services function blocks

Any registered Web service can be exported from System Manager for use in Activity Editor. The export process converts each method in the Web service to a function block, which is then placed in the My Library\Web Services folder. After Web service methods have been exported and converted to function blocks, they can be used in activity definitions. Use them the same way as other function blocks—drag and drop them on the editing canvas and specify the required inputs and outputs.

To export a Web service and convert its methods to function blocks, do the following:

1. Ensure System Manager is open.
2. Locate the Web service you want to export from the appropriate integration component library (ICL).
3. Right-click the name of the Web service to bring up the context menu.
4. Click **Export to Activity Editor** from the context menu. The Export to Activity Editor dialog box opens.
5. Select the check box next to the Web service or services you want to export, and then click **Finish**. The Web service methods are converted to function blocks and exported to Activity Editor.
6. If you have an open Activity Editor session, you must now exit and restart Activity Editor to make the new function blocks accessible.

Connection links

Function blocks are connected by *connection links*. Connection links define the flow of activity between the various components in the activity definition. They connect the output port of one function block to the input port of another function block.

Note: Output ports can connect to multiple connection links, but input ports can accept only a single connection link.

Perform the following steps to add a connection link between two function blocks:

- Click and hold down the left mouse button on the output port of the first function block (Function Block A).
- While holding down the left mouse button, move the cursor onto the input port of the second function block (Function Block B).
- Release the mouse button. The connection link is placed between the two function blocks. It is represented graphically by a right-angled line between the two components.

If an input port already has an existing connection link the newer connection link replaces it.

New Constant function block

Activity Editor has a New Constant function that you can drag and drop onto the editing canvas to define a constant value that you set and use as input to other function blocks.

The New Constant function block is located in the System folder in the Library and Content windows. Figure 54 illustrates what the New Constant function block looks like when it is dropped onto the editing canvas.



Figure 54. New Constant function block

The text edit box is displayed on top of the function block so you can enter the constant's value. (If you need to edit the value, click inside the constant function block and enter the new value.) Note that the constant contains a single output port.

Note: The Constant function block is the only activity definition component that accepts only a single line for the value. The constant is translated into a Java String object, and the system cannot translate a multi-line constant value. If it is absolutely necessary to have multi-line input, use the "`\n`" programming convention to separate lines in the constant. (For example the value "`line1\nline2`" indicates that the system must output the value in two lines.)

Tags for activity definitions

The System folder (located in the Library and Content windows) contains function blocks for adding comment, description, label, and to-do tags to the activity definition. These tags help identify each activity or subactivity, or serve as a reminder of something that must be done. You drag and drop these function blocks onto the editing canvas as you would any other function block. However, there are no input and output ports.

To edit a new tag, single-click in the center of the tag. The cursor changes to an I-beam, and you can enter your text. The tags automatically wrap lines of text that are too long. You can also press Enter to type text on a new line.

If you want to resize a tag, left-click the lower right-hand corner of the tag, and then hold down the mouse while dragging the tag to the desired size. Note that the tags do have a minimum size requirement, and cannot be resized smaller than that minimum size.

Component groups

A set of function blocks on the editing canvas can be grouped together and saved for later reuse in another activity definition. In effect, this saved component group acts as a function block.

After you have created the desired activity flow on the editing canvas, perform the following steps to save all or part of the flow as a reusable component group:

1. Select the function blocks you want to group together. Hold down the Ctrl key to select multiple function blocks.
2. Right-click the editing canvas to open the context menu.
3. Click Add to my Collection. The Add to My Collection dialog box is displayed.
4. Enter a name and (optionally) a description for the component group you are creating.
5. Select the icon you want to use to represent the component group, and then click OK.

The new component group icon is added to the My Collection folder in the Library and Content windows. You can drag and drop the icon onto the editing canvas for any activity definition within your collaboration scenario.

Supported function blocks

Activity Editor’s function blocks are organized under the General folder in the Library window, and in the corresponding folders in the Content window. Table 47 describes how the function blocks are organized.

Table 47. Organization of function blocks

Function block folder	Description	For more information
General\APIs\Business Object	Function blocks for working with business objects.	Chapter 11, “Business object function blocks,” on page 237
General\APIs\Business Object\Array	Function blocks for working with Java arrays in the BusObj class.	Chapter 11, “Business object function blocks,” on page 237
General\APIs\Business Object\Constants	Function blocks for working with Java constants in the BusObj class.	Chapter 11, “Business object function blocks,” on page 237
General\APIs\Business Object Array	Function blocks for working with business object arrays.	Chapter 12, “Business object array function blocks,” on page 261
General\APIs\Database Connection	Function blocks for creating and maintaining a database connection.	Chapter 14, “Database connection function blocks,” on page 285
General\APIs\Identity Relationship	Function blocks for working with identity relationships.	<i>Map Development Guide</i>
General\APIs\Maps	Function blocks for querying and setting runtime values needed for map execution.	<i>Map Development Guide</i>
General\APIs\Maps\Constants	Function block constants.	<i>Map Development Guide</i>
General\APIs\Maps\Exception	Function blocks for creating new exception objects in a map.	<i>Map Development Guide</i>
General\APIs\Participant	Function blocks for setting and retrieving values for participants in an identity relationships.	<i>Map Development Guide</i>

Table 47. Organization of function blocks (continued)

Function block folder	Description	For more information
General\APIs\Participant\Array	Function blocks for creating and working with participant arrays.	<i>Map Development Guide</i>
General\APIs\Participant\Constants	Function block constants for use with participants.	<i>Map Development Guide</i>
General\APIs\Relationship	Function blocks for manipulating runtime instances of relationships.	<i>Map Development Guide</i>
General\Date	Function blocks for working with dates.	Chapter 18, "Date function blocks," on page 311
General\Date\Formats	Function blocks for specifying different date formats.	Chapter 18, "Date function blocks," on page 311
General\Logging and Tracing	Function blocks for handling log and trace messages.	Chapter 19, "Logging and tracing function blocks," on page 317
General\Logging and Tracing\Log Error	Function blocks for formatting error messages.	Chapter 19, "Logging and tracing function blocks," on page 317
General\Logging and Tracing\Log Information	Function blocks for formatting informational messages.	Chapter 19, "Logging and tracing function blocks," on page 317
General\Logging and Tracing\Log Warning	Function blocks for formatting warning messages.	Chapter 19, "Logging and tracing function blocks," on page 317
General\Logging and Tracing\Trace	Function blocks for formatting trace messages.	Chapter 19, "Logging and tracing function blocks," on page 317
General\Mapping	Function blocks for executing maps within a specified context.	<i>Map Development Guide</i>
General\Math	Function blocks for basic mathematical tasks.	<i>Map Development Guide</i>
General\Properties	Function blocks for retrieving configuration property values.	<i>Map Development Guide</i>
General\Relationship	Function blocks for maintaining and querying identity relationships.	<i>Map Development Guide</i>
General\String	Function blocks for manipulating String objects.	Chapter 20, "String function blocks," on page 325
General\Utilities	Function blocks for throwing and catching exceptions, as well as looping, moving attributes, and setting conditions.	Chapter 21, "Utilities function blocks," on page 333
General\Utilities\Vector	Function blocks for working with Vector objects.	Chapter 21, "Utilities function blocks," on page 333

Example: Changing a date format

This example illustrates how to use Activity Editor to change a source attribute's date format and assign the reformatted value to a destination attribute. In this example, the source attribute is QuoteSchedule.ExpireDate and the destination attribute is Invoice.PostingDate. The original date format is yyyyMMdd, and the updated date format is yyyy-MM-dd. This example assumes that the business objects and attributes have already been created and declared in the collaboration template scenario.

The following steps are required to change the source attribute's date format and then assign it to the destination attribute:

1. Ensure that Activity Editor is open.
2. Drag the QuoteSchedule.ExpireDate variable function block onto the editing canvas and drop it. (The function blocks that represent the business objects, attributes, and variables available in a scenario are located in the Variables folder of the Library and Content windows.)
3. Drag and drop the Format Change function block onto the editing canvas to the right of the QuoteSchedule.ExpireDate function block, as shown in Figure 55.

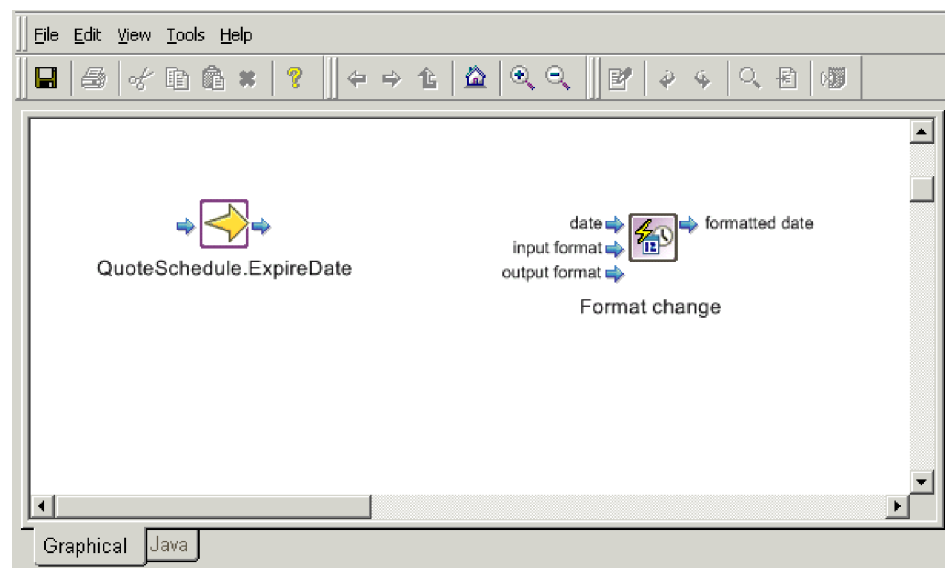


Figure 55. Placing the Format Change function block

4. Place a connection link between the output port of the QuoteSchedule.ExpireDate function block and the Date input of the Format Change function block.
5. Drag and drop the yyyyMMdd function block constant onto the editing canvas, placing it underneath the QuoteSchedule.ExpireDate and Format Change function blocks. This function block represents the current format of the QuoteSchedule.ExpireDate attribute.
6. Place a connection link between the output port of the yyyyMMdd function block and the Input Format input of the Format Change function block, as shown in Figure 58 on page 161.

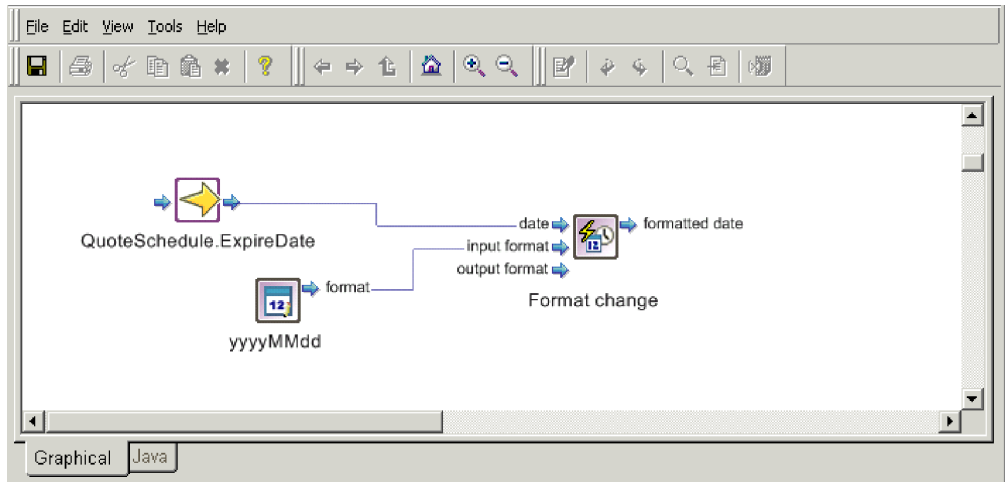


Figure 56. Specifying the input date format

7. Drag and drop the yyyy-MM-dd function block constant onto the editing canvas, placing it near the yyyyMMdd function block. This function block represents the new format of the QuoteSchedule.ExpireDate attribute.
8. Place a connection link between the output port of the yyyy-MM-dd function block and the Output Format input of the Format Change function block, as shown in Figure 58 on page 161.

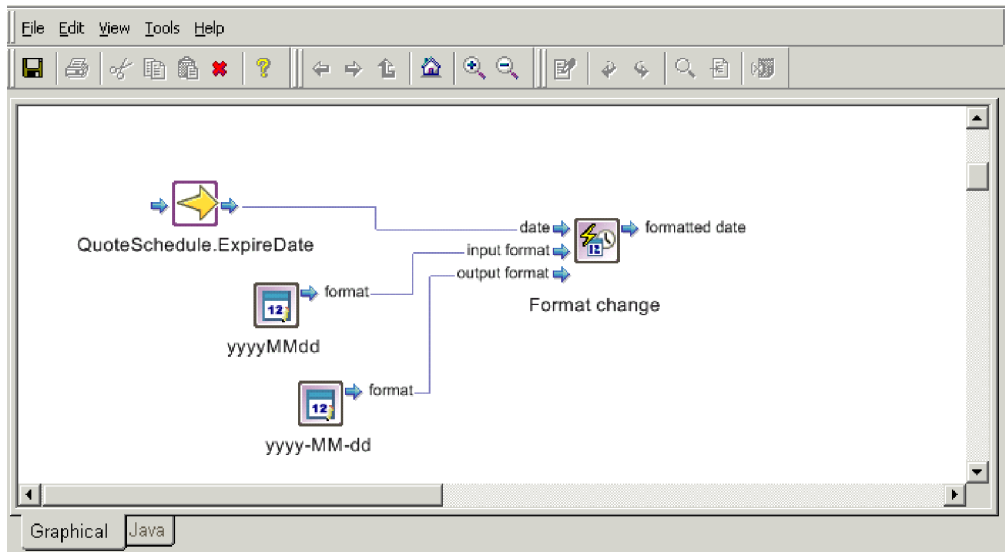


Figure 57. Specifying the output date format

9. Drag and drop the Invoice.PostingDate function block to the editing canvas; this is the destination attribute. Place it to the right of the Format Change function block.
10. To assign the output of the Format Change function block to the Invoice.PostingDate attribute, place a connection link between the Format Change function block output port and the Invoice.PostingDate input port as shown in Figure 58 on page 161.

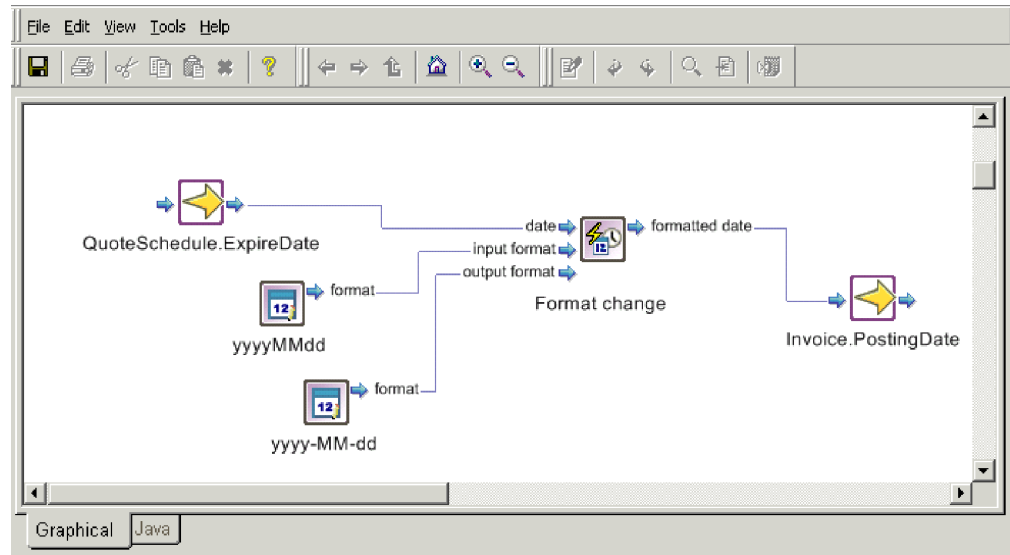


Figure 58. Assigning output to the destination attribute

11. Save the activity definition by clicking File → Save.

Example: Creating a duplicate business object

The following example demonstrates how to duplicate a business object. In this example, the original object is the triggering business object (triggeringBusObj), and the duplicate is going to be the business object for a port variable called inPort (inPortBusObj). Both business objects are located in the Variables folder of the Library and Content windows.

To create the duplicate business object for this example:

1. Ensure that Activity Editor is open.
2. Drag and drop the triggeringBusObj variable onto the editing canvas.
3. Drag and drop the Duplicate function block onto the canvas, placing it to the right of the triggeringBusObj function block.

Note: The Duplicate function block is located in the General\APIs\Business Object folder in the Library and Content windows.

4. Place a connection link between the output port of the triggeringBusObj variable and the Original input of the Duplicate function block.
5. Drag and drop the inPortBusObj variable onto the canvas, placing it to the right of the Duplicate function block.
6. To assign the value of the original business object to the new business object, place a connection link between the duplicate output of the Duplicate function block and the input for the inPortBusObj variable.



Figure 59. Duplicating a business object

7. Save the activity definition by clicking File—>Save.

Importing Java packages and other custom code

Process Designer provides a way to import Java packages and other custom code as described in the following sections.

Importing Jar libraries as activity function blocks

Besides using the standard function blocks that Activity Editor provides, Process Designer allows you to import your own Java library for use as function blocks in the Activity Editor. Importing custom Jar libraries into activity settings will enable any public methods in the Jar library to be used as function blocks in the Activity Editor.

Steps for importing Jar libraries as activity function blocks

Before you begin: Export your Java classes into a .jar file.

To import a Jar library into Activity Editor:

1. Open the Activity Settings view in System Manager by clicking Window > Show View > Other and selecting Activity Settings from the System Manager category.
2. Right-click BuildBlock Libraries and select Add Library. Figure 60 shows the Activity Settings view for adding a custom Jar library.

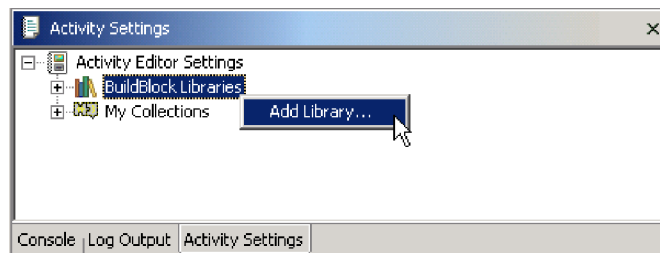


Figure 60. Activity Settings view

3. In the Open File dialog box, navigate to your custom .jar files and select Open. System Manager will try to import your custom .jar file for use as function blocks in Activity Editor. If the file is imported properly, the name of your custom .jar file will appear under BuildBlock Libraries in the Activity Settings view.

Tip: After importing your custom .jar files into Activity Settings, when your maps and collaboration template compile in the System Manager, the custom .jar file will automatically be included into the compile CLASSPATH. To prepare the InterChange Server for compilation, make sure that its CLASSPATH includes your custom.jar file. For information on setting up InterChange Server for importing your custom .jar files, see “Importing third-party classes to InterChange Server” on page 165.

4. Restart the Process Designer.

Rule: After you change any settings in the Activity Settings view, you must restart the Process Designer for the change to take effect in the Activity Editor.

Result: When you open the Activity Editor, the custom Jar library will be listed in the Library window under My Library in Activity Editor. By default, available custom function blocks are listed according to their package structure. You can use them in an activity the same way as you use standard function blocks.

Customizing display settings of custom Jar libraries

You can customize the display settings of function blocks imported into the Activity Editor, such as its name and icon, by changing the custom Jar library’s properties. To change the Jar library’s properties:

- Display the Properties window for the custom Jar library by right-clicking on your custom Jar library listed under BuildBlock Libraries in the Activity Settings view in System Manager.

Result: When the Properties window for the custom Jar library is opened, it will list the available function blocks in this custom library in a tree structure on the right-hand side of the dialog. The available function blocks are listed as child nodes under the Java class and package that contain them.

For the Java package and classes, you can customize the display name of the entry and whether Activity Editor should display this Java package/class in the My Library tree structure by changing the check box “Hide level in tree display.” If this option is enabled, Activity Editor will not display this entry in the My Library subtree. This option is usually useful when the Java methods in your custom Jar library are in a Java class that is in a package many levels deep. Enabling this option better organizes your My Library subtree in Activity Editor.

Figure 61 shows the dialog for customizing the Jar library display.

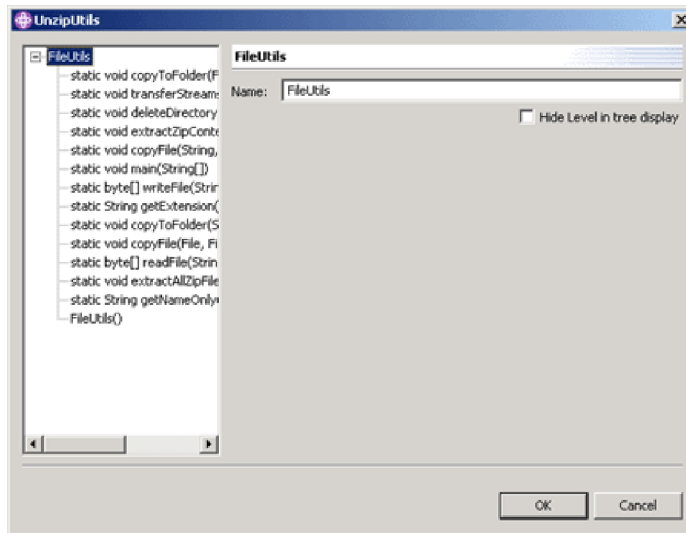


Figure 61. Properties dialog for customizing Jar library display

For those Java methods used as function blocks in the Activity Editor, you can specify the function block display name, description, icon, and parameter's display name in the Properties window. When you import an icon for the function block, the icon that you select will be copied into the Activity Settings folder and will be available for other function blocks in the same package to use.

Recommendation: When importing an icon for your function block to use, the icon should be 32 pixels by 32 pixels in size and should be in .bmp format. The color depth of the icon can be up to 24-bit.

Figure 62 shows the Properties dialog for customizing the Jar library function block display.

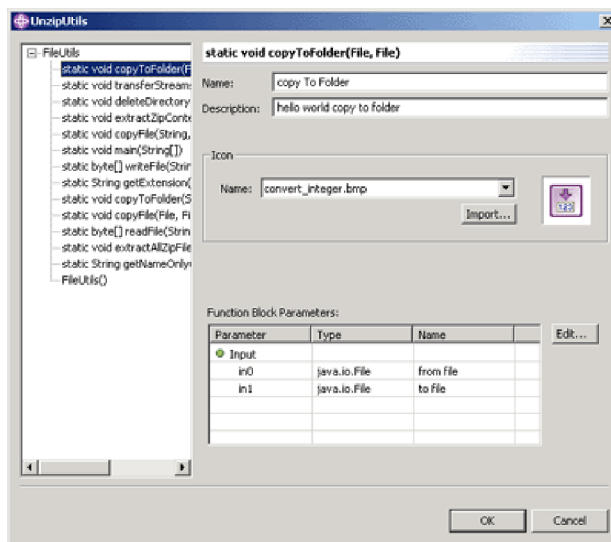


Figure 62. Properties dialog for customizing Jar library function block display

Rule: After changing any settings in the Activity Settings view, you must restart Process Designer for the change to take effect in the Activity Editor.

Importing third-party classes to InterChange Server

If the imported classes are in a third-party package rather than in the JDK, to set up the server compile, you must add them to the path of the imported classes in the JCLASSES variable.

Recommendation: Use some mechanism to differentiate those classes in JCLASSES that are standard from those that are custom.

Example: You can create a new variable to hold only those custom classes and append this new variable to JCLASSES, by doing the following:

1. Create a new property, such as one called DEPENDENCIES.
2. Place the CwMacroUtils.jar in its own directory.

Example: Create a dependencies directory below the product directory and place the jar file in it.

3. Add the dependencies directory to the file used to start InterChange Server (by default, start_server.bat or CWSharedEnv.sh), which is located in the bin directory below the product directory. For example, add the following entry for UNIX:

```
set DEPENDENCIES=$ProductDir/dependencies/CwMacroUtils.jar
```

Add the following entry for Windows:

```
set DEPENDENCIES="%ProductDirS%\dependencies\CwMacroUtils.jar
```

4. Add DEPENDENCIES to the JCLASSES entry:

For UNIX, add:

```
set JCLASSES=$JCLASSES:ExistingJarFiles:$DEPENDENCIES
```

For Windows, add:

```
set JCLASSES=ExistingJarFiles;%DEPENDENCIES%
```

5. In each map that uses the classes, include the *PackageName.ClassName* specified in the CwMacroUtils.jar file.
6. Restart InterChange Server to make the methods available to the maps.

Note: Be sure you have modified the Compiler Classpath preference for compiling the maps and collaboration templates from System Manager. To modify the Compiler Classpath preference:

1. In System Manager, select Window > Preferences to open the Preference dialog.
2. Expand System Manager Preferences, and select Compiler.
3. In the Compiler preference page, click New and select the .jar file to be included in the compile CLASSPATH for maps and collaboration templates.

Guidelines: When importing a custom class, you may get an error message indicating that the software could not find the custom class. If this occurs:

- Check that the custom class is part of a package. It is good programming practice for custom classes to be placed in a package. Make sure that the custom class code includes a correct package

statement and that it is placed at the beginning of the source file, prior to any class or interface declarations.

- Verify that the import statement is correct in the map definition code. The import statement must reference the correct package name; it may further specify the name of the custom class or it may reference all classes in the package.

Example: If your package name is `COM.acme.graphics` and the custom class is `Rectangle`, you can import the entire package:

```
import COM.acme.graphics.*;
```

Or, you can import just the `Rectangle` custom class:

```
import COM.acme.graphics.Rectangle;
```

- Be sure that you have updated the `CLASSPATH` environment variable to include the path to the package containing the custom class, or to the custom class itself if it is not in a package.

Example: When importing a custom class, you might create a folder called `%ProductDir%\lib\com\<ProductDir>\package`, where `package` is the name of your package. Then, place your custom class file under the folder you just created. Finally, in the `CLASSPATH` variable in the `start_server.bat` file, include the path `%ProductDir%\lib`.

Chapter 7. Handling exceptions

An *exception* represents an error situation that, if not handled explicitly within the activity diagram, can stop execution of the collaboration. The goal of exception handling is to ensure the following:

- If possible, the error condition that caused the exception is corrected or reduced in scope so that collaboration can continue execution.
- If the error condition cannot be corrected and scenario must end unsuccessfully, collaboration execution must terminate. In this case, the collaboration should try to provide as much information as possible about the cause of the error condition. This information helps the administrator determine how to fix this instance of the error and to prevent future occurrences of this error.

Therefore, it is important to understand how exceptions are handled, both by your collaboration template and by the collaboration runtime environment. This section provides the following information about handling exceptions:

- “What is a collaboration exception?”
- “How exceptions are processed” on page 168
- “How to handle exceptions” on page 171

What is a collaboration exception?

The Collaboration API provides an *exception object* to represent an exception that occurs in a collaboration. As Table 48 shows, this exception object contains information about the condition that caused the exception.

Table 48. The CollaborationException exception object

Exception Type
Exception Subtype
Message
Message Number

This exception object is an instance of the CollaborationException class, which is an extension of the Java Exception class. Table 49 shows the accessor methods that the CollaborationException class provides to obtain information in the exception object.

Table 49. Information in the exception object

Member	Accessor method
Exception type	getType()
Exception subtype	getSubType()
Message text	getMessage(), toString()
Message number	getMsgNumber()

Note: When an exception occurs, the collaboration runtime environment populates a system variable called `currentException` with information about the exception. The `currentException` variable is an instance of the CollaborationException class. Therefore, you can use the methods in Table 49 to obtain exception information from the `currentException` variable.

To identify the cause of the collaboration exception, the exception object includes one of the *exception types* listed in Table 50. Exception types are string values for which Java static constants have been declared.

Table 50. Exception types

Exception-type constant	Description
AnyException	Any type of exception. If there are two exception branches—one that tests for a specific type of exception and one that tests for AnyException—the branch that tests for the specific type of exception is checked first. If the current exception does not match the specific exception, the branch that tests for AnyException is processed next.
AttributeException	Attribute access problem, for example, the collaboration called getDouble() on a String attribute or called getString() on a nonexistent attribute.
JavaException	Problem with Java code that is not part of the collaboration API.
ObjectException	Invalid business object passed to a method.
OperationException	Service call was improperly set up and could not be sent.
ServiceCallException	Service call failed for reasons outside the collaboration. For example, a connector or application is unavailable, or there is a network outage.
SystemException	Any internal error within the WebSphere InterChange Server system.
TimeoutException	Synchronous or asynchronous inbound service call timed out.
TransactionException	Error related to the transactional behavior of a transactional collaboration. For example, rollback failed or the collaboration could not apply compensation.

Note: When you define an exception branch in a decision node, you specify the exception type to check for in the condition of the exception branch. For more information, see “Catching the exception” on page 172.

Some of these exception types have numerous situations that can cause them. For such exception types, the exception object often includes an *exception subtype*, which provides additional information about the cause of the exception. The two main exception types that use exception subtypes are JavaException and ServiceCallException. For more information, see the description of “getSubType()” on page 424.

How exceptions are processed

As a collaboration executes, it can be in one of the following two *execution states*:

- The Normal state indicates either of the following conditions:
 - No exception has occurred.
 - An exception has occurred but the collaboration template has caught the exception.

- The Exception state indicates that an exception has occurred and has *not* been handled within the collaboration template. A collaboration enters an exception state when either of the following conditions occurs:
 - A collaboration exception occurs, such as one of the following conditions:
 - A service call fails; that is, a collaboration exception with an exception type of `ServiceCallException` occurs.
 - A Java exception occurs; that is, a collaboration exception with an exception type of `JavaException` occurs.
 - The collaboration template calls the `raiseException()` method, raising a collaboration exception with any valid exception type.

The collaboration can switch back and forth between these two states during its execution. It enters the Exception state when the exception occurs or `raiseException()` executes. It returns to the Normal state when the collaboration template catches the exception with an exception branch. Regardless of the exception handling that an activity diagram does (or does not) perform, the collaboration runtime environment continues execution of the diagram's logic after an exception occurs. This logic eventually ends in either an End Success or End Failure node. The collaboration runtime environment uses the collaboration's execution state to determine whether to create an unresolved flow once the collaboration ends. For more information on terminating nodes, see "Terminating the execution path" on page 143. For more information on unresolved flows, see "Processing the Exception state."

Processing the Normal state

While the collaboration runtime environment is successfully executing a collaboration (as defined by the logic in the activity diagrams), the collaboration's execution is in the Normal state. Possible ways to end the execution path include:

- An End Success node

The collaboration runtime environment stops execution of the current diagram and passes control to the next higher level of execution:

- If the End Success node terminates the main activity diagram, the collaboration runtime environment ends the collaboration. When the collaboration's execution is in the Normal state, the collaboration runtime environment does *not* create an unresolved flow.
- If the End Success node terminates a subdiagram or iterator, the collaboration runtime environment takes the following steps:
 - End the current level of execution and pass control to the parent diagram.
 - Continue execution of the parent diagram as defined by its logic. However, when the collaboration's execution is in the Normal state, the collaboration runtime environment does *not* check for any exception branches.

- An End Failure node

The collaboration runtime environment stops execution of the collaboration and performs the steps for a collaboration in the Normal state (see "Terminating in failure" on page 144).

Processing the Exception state

When the collaboration's execution enters the Exception state, the collaboration runtime environment does *not* stop execution. Instead, it continues execution as defined by the logic in the activity diagram, just as it does for execution in the Normal state. Possible ways to end this execution path include:

- An End Success node

The collaboration runtime environment stops execution of the current diagram and passes control to the next higher level of execution, which can be:

- If the End Success node terminates the main diagram, the next higher level of execution is the collaboration runtime environment. For more information, see “Successfully ending the main diagram.”
- If the End Success node terminates a subdiagram or iterator, the next higher level of execution is the parent diagram. For more information, see “Successfully ending a subdiagram or iterator” on page 171.

- An End Failure node

The collaboration runtime environment stops execution of the collaboration. When the collaboration’s execution is in the Exception state, the collaboration runtime environment next handles the exception. For exception-handling steps, see “Successfully ending the main diagram.”

Successfully ending the main diagram

When the End Success node terminates the main diagram, the collaboration runtime environment ends the collaboration. If the collaboration’s execution is in the Exception state, the runtime environment performs the following steps to handle the exception:

1. Log an error to the collaboration’s log destination, which can be standard output (STDOUT) or a log file, depending on how InterChange Server’s log destination is configured.

- If the collaboration is *not* transactional, the collaboration runtime environment logs an error.
- If the collaboration is transactional, the collaboration runtime environment rolls it back, executing the collaboration’s compensation steps.

If the exception occurs during rollback, the collaboration runtime environment terminates the collaboration and logs the error. At this point, the collaboration object is in an “in-doubt” state. An administrator can manually resolve the collaboration object’s transactional status by executing or discarding the remaining compensation steps.

2. Create an *unresolved flow* for the unsuccessful collaboration.

When a collaboration ends with its execution in the Exception state, it leaves behind an unresolved flow, which includes:

- A **failed event**, which is the original event (business object and verb) that triggered the unsuccessful collaboration
- An exception message to describe the cause of the failure

The collaboration runtime environment sends this unresolved flow to InterChange Server’s event resubmission queue, where the server administrator can analyze and assess it for possible resubmission. The Flow Manager tool provides the administrator with access to the event resubmission queue. The administrator can examine information about unresolved flows, such as the name of the collaboration that terminated and a message that describes the error condition.

Note: For more information on use of the Flow Manager, see the *System Administration Guide*.

3. Handle the exception, save the failed event and return success to the main scenario. At some future point, the administrator can resubmit the failed events to the collaboration for processing. See “Saving a failed event” on page 75.

By default, the collaboration runtime environment associates a very simple exception message with an unresolved flow:

Scenario failed.

This default exception message does *not* provide the administrator with much information with which to troubleshoot the cause of the unresolved flow. However, if you code the collaboration template to raise the exception, you can provide an exception message with more information about the actual error condition that occurred. When the collaboration runtime environment handles the exception, it can associate this more detailed exception message with the unresolved flow. For more information, see “Raising the exception” on page 174.

Successfully ending a subdiagram or iterator

When the End Success node terminates a subdiagram (or iterator) and the collaboration’s execution is in the Exception state, the collaboration runtime environment takes the following steps:

1. Pass control to the parent diagram. The parent diagram is the diagram that includes the subdiagram (or iterator) node.
2. Check for any exception branches in the parent diagram’s decision node that connect the subdiagram (or iterator) with an exception-handling node. Take one of the following actions:
 - If an exception branch exists that catches the current exception, the collaboration runtime environment passes control to the exception-handling node to which the exception branch points. When this exception-handling node completes, execution continues as defined by the branch of the activity diagram that contains the exception-handling node.

Note: Once an exception branch executes, the collaboration’s execution changes to the Normal state. Therefore, unless the collaboration template raises the exception somewhere in the execution path, the collaboration runtime environment does *not* create an unresolved flow for the collaboration. For information on how to implement exception-handling code, see “How to handle exceptions.”

- If no exception branch catches the current exception, the collaboration runtime environment continues execution of the parent diagram as defined by its logic. However, the collaboration’s execution is still in the Exception state. Unless some other level of execution catches the exception, the collaboration runtime environment creates an unresolved flow for the collaboration when the collaboration ends.

The collaboration runtime environment continues execution of the parent diagram’s logic until that diagram terminates in either an End Success or End Failure. As long as the collaboration execution remains in the Exception state, the runtime environment handles the exception when the collaboration ends. As long as each level of execution ends with an End Success, execution passes to the next higher level until it reaches the main diagram. Unless the main diagram catches the exception, this collaboration terminates and control passes to the collaboration runtime environment.

How to handle exceptions

There are two categories of exceptions that a collaboration can handle:

- Business process exceptions

Business process exceptions arise from code that uses the Collaboration API methods. For example, a business process exception can occur when the scenario sets the value of a business object attribute, sends a service call request to a

connector, and so on. For information on how to handle certain service-call exceptions, see “Handling particular service-call exceptions” on page 178.

- Native Java exceptions

Java exceptions result from your own code that uses native Java methods. When such an exception occurs, you can raise a collaboration exception whose exception type is `JavaException` and whose exception subtype contains the particular Java exception. The collaboration runtime environment catches and handles the Java exceptions arising from its own code.

When an exception occurs, the exception handling at a given level of the activity-diagram hierarchy can handle this exception in one of the following ways:

- Do *not* catch the exception at the current level of execution.
- Catch the exception with an exception branch in a decision node.

Not catching the exception

If the activity diagram does *not* explicitly catch the exception with an exception branch, the collaboration’s execution remains in the Exception state (which it entered when the exception occurred). The collaboration runtime environment does *not* stop execution in the diagram when the exception occurs. Instead, execution continues according to the logic of the activity diagram, ending in either an End Success or an End Failure node:

- If the execution path ends in End Failure, the collaboration runtime environment terminates the collaboration and creates an unresolved flow.

Therefore, if you want to catch an exception in the subdiagram but also want to traverse the End Failure node, you must make sure that the code catches the exception within the subdiagram (before the End Failure node).

- If the execution path ends in End Success, the collaboration passes control to the next higher level.

In a hierarchical activity diagram, if you do not use an exception branch to catch an exception at one level of execution, you can use End Success to pass control to the next higher level diagram. In this higher-level diagram, you can catch the event and either handle the exception or raise it to the next higher level.

If the collaboration template has not caught the exception *anywhere* in its execution path, its execution remains in the Exception state. In this case, the collaboration runtime environment still handles the exception, as described in “Processing the Exception state” on page 169. Because the collaboration template has never caught the exception, the collaboration runtime environment must include its own default exception message (Scenario failed.) with the unresolved flow.

Catching the exception

A collaboration template can incorporate exception handling in its activity diagram with *exception branches*, which are branches within a decision node whose branch type is set to Exception. The decision node connects the action symbol to its possible decision outcomes. An exception branch routes the action symbol in which the exception occurs to the action symbol in which the exception is handled. An exception branch includes an exception condition, which specifies the exception type that the exception branch catches. Table 50 on page 168 lists the collaboration exception types that you can choose from when you define the exception condition.

Note: For more information on how to add an exception branch to an activity diagram, see “Defining an exception branch” on page 123.

When an exception occurs, the collaboration runtime environment populates the `currentException` system variable. To determine whether to follow the exception branch, the collaboration runtime environment evaluates the exception condition of the exception branch by comparing the exception type in the exception branch's condition with the exception type in the `currentException` system variable:

- If these exception types match, the exception condition is *true* and the activity diagram *catches* the exception.

The collaboration runtime environment changes the collaboration's execution goes to the Normal state and passes control to the action symbol to which the exception branch points. This action symbol can contain the exception-handling code to handle the exception type specified in the exception condition. This code can access the `currentException` system variable to obtain exception information.

- If these exception types do *not* match, the exception condition is false and the activity diagram *does not catch* the exception.

Execution passes to the default branch of the decision node (if one exists) and then on to the next action symbol in the logic of the activity diagram. Unless the exception is handled in the default branch, this situation means that the exception is *not* handled at this level of the activity diagram. It also means that the collaboration's execution remains in the Exception state.

Note: The collaboration runtime environment checks for exception branches *only* when the collaboration execution is in the Exception state.

A given decision node can have a maximum of seven branches. Therefore, it can provide exception-handling for many exception types. Each exception branch can specify a different exception type in its exception condition and can point to an exception-handling node for that exception type. Alternatively, you can handle *all* exception types in a single exception branch that has the `AnyException` exception type in its exception condition.

Once the collaboration template has caught the exception and execution passes to the exception-handling node, the collaboration template can handle it in the following ways:

- Proceed with the logic of the scenario to successful or unsuccessful completion.
- Raise the exception to the next higher level in the activity diagram.

Proceeding with scenario logic

To proceed with the logic of the scenario, you take the following steps in the exception-handling node:

1. Process the exception in a manner that does *not* involve raising the exception.

Within the node to which the exception branch points, you can include code that processes the exception. Table 51 on page 174 lists some possible processing steps.

2. End the execution path in the activity node with either an End Success or an End Failure node.

As long as the exception-handling node does *not* raise the exception (with the `raiseException()` method), the collaboration's execution remains in the Normal state. Therefore, the collaboration runtime environment does *not* create an unresolved flow when collaboration execution completes. For more information on how the collaboration runtime environment processes these termination nodes, see "Processing the Normal state" on page 169.

Table 51 lists some of the possible processing steps that the exception-handling node can perform. None of these steps changes the collaboration's execution state. Therefore, the collaboration's execution remains in the Normal state.

Table 51. Possible processing steps for exception handling

Exception-handling step	Methods	For more information
Log a message in the collaboration's log destination.	logError(), logWarning(), logInfo()	"Logging messages" on page 193
Obtain information about the exception.	Methods of the CollaborationException class.	Table 49 on page 167

For example, the logError() method logs errors to the collaboration's log destination. This destination can be standard output (STDOUT) or to a log file, if configured to do so. This method also sends the error message to an email recipient. The collaboration template can use this method to record errors that the administrator can examine. The following code fragment extracts exception information from the currentException variable with the getMessage() and getMsgNumber() methods of the CollaborationException class. It then uses this information to format the error to send to the collaboration's log destination with a call to logError().

```
// extract exception information
sMessage = currentException.getMessage();
iMsgNumber = currentException.getMsgNumber();

// log message and send email (if configured)
logError(iMsgNumber, sMessage, ...);
```

For more information, see the description of the logError() method in "logError(), logInfo(), logWarning()" on page 353. Keep in mind that merely sending an error to the log destination is often insufficient for associating a clear exception message with the unresolved flow. For example, suppose that an exception occurs, an exception branch catches it, and the exception-handling node simply logs the error and ends in failure. In this case, the unresolved flow for the unsuccessful collaboration includes the failed event but its exception message is just the collaboration runtime environment's default message (Scenario failed.).

Raising the exception

The raiseException() method raises a collaboration exception to the next higher level of execution. When the collaboration runtime environment executes the raiseException() call, it changes the collaboration's execution state to Exception, then proceeds with the logic of the activity diagram. To raise the exception to the next higher level in the activity diagram, you take the following steps in the exception-handling node:

1. Acquire exception information from the current exception to include in the raised exception.

Within the exception-handling node, you can use methods of the CollaborationException class to extract exception information from the currentException system variable.

Note: It is important to extract the message from the currentException variable so that it can be included in the raised exception. By doing so, this message can be available to the collaboration runtime environment when it associates an exception message with the unresolved flow.

2. Include a call to the `raiseException()` method to generate the exception to raise.

When the collaboration runtime environment executes a call to `raiseException()`, it changes the collaboration's execution to the Exception state. The `raiseException()` call provides the exception to raise to the next higher level of execution.

3. End the execution path for the branch that contains the exception-handling code with either an End Success or an End Failure node.
 - If you end the execution path in End Success, you raise the exception to the next higher level of execution, where it can be caught or raised to the next higher level. When you use this method at each execution level, the collaboration code can raise trapped exceptions to the top-level diagram, which can make the final determination for error handling.
 - If you end the execution path in End Failure, you raise the exception to the collaboration runtime environment, which includes the exception message as part of the unresolved flow.

Because you have raised the exception, the collaboration execution is in the Exception state. As long as each level of execution raises the exception (with `raiseException()`), the collaboration's execution remains in the Exception state. Therefore, the collaboration runtime environment creates an unresolved flow when collaboration execution completes. For information on how the collaboration runtime environment processes these termination nodes, see "Processing the Exception state" on page 169.

To understand how you can use the `raiseException()` and `logError()` methods in collaboration templates to handle exceptions, consider the following example. Suppose that a collaboration's main diagram calls `subdiagramA`, which in turn calls `subdiagramB`. `SubdiagramB` makes a service call, which could result in an exception. Therefore, this subdiagram contains an action node that invokes a service call. This action node connects to a decision node with an exception branch that checks for service-call exceptions. If a service-call exception occurs, the exception branch connects to an action node with the exception-handling code.

When an exception occurs, the collaboration runtime environment changes the collaboration's execution to the Exception state and evaluates the exception condition of any exception branches that are associated with the service call in `subdiagramB`. If an exception branch's condition evaluates to true, the exception branch catches the exception and control passes to the exception-handling node to which this exception branch points. Once the exception branch catches the exception, the collaboration's execution returns to the Normal state.

Figure 63 shows the code fragment that performs the exception handling for a service-call exception in `subdiagramB`.

```
// exception handling in subdiagramB
sMessage = currentException.getMessage();
sType = currentException.getType();

// raise the exception to subdiagramA
raiseException(sType, 2345, parameter1, parameter2, sMessage);
}
```

Figure 63. Handling the service-call exception in subdiagramB

The code in this exception-handling node takes the following steps:

1. Check the `currentException` system variable for information about the exception.

The code obtains the exception message and exception type from `currentException` and saves them in two string variables (`sMessage` and `sType`, respectively).

2. Raise the exception to the parent diagram (in this case, `subdiagramA`).

Once it gathers the exception information, the code calls the `raiseException()` method to raise the exception to `subdiagramA`. This form of the `raiseException()` method receives the exception information as an exception type and an error message (2345) with its three message parameters. These message parameters include the exception message that the code obtained from the `currentException` system variable. The `raiseException()` call then creates an exception that contains this exception information. It also changes the collaboration execution to the Exception state.

Notes:

- a. The number of message parameters that `raiseException()` provides for a message depends on the particular format of the message in the collaboration message file.
- b. You can specify the same exception types when you raise an exception in the exception-handling code with the `raiseException()` method as you can when you define the exception condition in the exception branch. For a list of exception types, see Table 50 on page 168.

After `raiseException()` executes, the action that the collaboration runtime environment takes depends on the termination node that ends the execution path. If the execution path terminates in an End Success node, the collaboration runtime environment takes the following steps:

- Pass control to `subdiagramA`.
- Check for a decision node with an exception branch that catches this exception (because the execution state is Exception). This decision node would connect to the node for `subdiagram B` and its exception branch would connect to the appropriate exception-handling node.

Note: If the execution path terminates in an End Failure node, the collaboration runtime environment terminates the entire collaboration. Because the execution state is Exception, the runtime environment creates an unresolved flow using the exception information in the exception that the `raiseException()` call raised.

If a decision node with exception branches exist in `subdiagramA`, the collaboration runtime environment evaluates each exception branch's condition. If this exception condition evaluates to true, `subdiagramA` catches the exception that `subdiagramB` raised. The collaboration execution changes to the Normal state and control passes to the exception-handling node, which handles the exception by raising it up to the parent diagram (the main diagram) with the following call to `raiseException()`:

```
// exception handling in subdiagramA: raise the exception to main diagram
raiseException(currentException);
```

This form of the `raiseException()` method just raises the exception object that it receives as an argument. It does *not* create an exception from information passed in. In this case, there is no need for `raiseException()` to create an exception because the exception-handling code in `subdiagramB` (Figure 63 on page 175) has already created the exception with the appropriate exception information. The exception that `subdiagramA` has in its `currentException` variable is the same

exception that subdiagramB raised. After the `raiseException()` completes, the collaboration execution continues according to the logic in subdiagramA. If this exception-handling branch of subdiagramA ends in End Success, the collaboration runtime environment terminates subdiagramA and passes control to its parent diagram, which is the main diagram. Therefore, the exception object that `raiseException()` generates (the `currentException` exception object in subdiagramA) is now raised to the main diagram.

Note: If this exception-handling branch of subdiagramA ended in End Failure, the collaboration ends and the exception object is raised to the collaboration runtime environment, which includes its exception message as part of the unresolved flow.

Collaboration execution is currently at the subdiagramA node in the main diagram. The collaboration execution is currently in the Exception state because the exception-handling node in subdiagramA called `raiseException()`. Therefore, the collaboration runtime environment checks in the main diagram for any exception branches that catch the raised exception. These exception branches would be within a decision node that connects the call to subdiagramA to one or more exception-handling action nodes. If an exception branch's condition evaluates to true, the main diagram catches the exception that subdiagramA raised. The collaboration's execution changes to the Normal state and control passes to the exception-handling node, which can take the appropriate high-level exception-handling steps.

As an example, suppose that this exception-handling node in the main diagram takes the following steps:

1. Verify whether the collaboration's `SEND_EMAIL` configuration property is set to either `all` or a comma-delimited list of message numbers.

All collaboration objects can indicate the recipients of email for errors that the `logError()` sends to the log destination. If a collaboration is based on `CollaborationFoundation`, it can take advantage of the additional functionality that the `SEND_EMAIL` collaboration property provides. If the collaboration object is configured to send email *and* `SEND_EMAIL` is set to either `all` or a list of message numbers, the collaboration sends email to the specified recipients when any error (`SEND_EMAIL` is `all`) or a specified error (`SEND_EMAIL` provides a message-number list) occurs.

If these conditions are met, the exception-handling node should call `logError()` to log the error and send email to the specified recipient. Therefore, the code must first retrieve the exception information to include in the error message from the current exception.

Note: The `SEND_EMAIL` configuration property is a feature of the `CollaborationFoundation`. If your collaboration is based on `CollaborationFoundation`, it can perform this checking of the `SEND_EMAIL` property. Otherwise, this configuration property is not defined.

2. Send the exception message to the collaboration's log destination and as an email message, if appropriate.

If the collaboration object is configured to send email, the `logError()` method automatically sends the error message to a specified email recipient. This branch uses the `logError()` method to send the exception to the collaboration's log destination (standard output or a log file).

The following code fragment in the exception-handling node of the main diagram performs these steps:

```

// exception handling in main diagram

// determine if SEND_EMAIL is set to "all" or a message-number list;
// if so, obtain exception information from the current exception
sMessage = currentException.getMessage();
imgNumber = currentException.getMsgNumber();

// log message and send email
logError(imgNumber, sMessage, ...);

// raise the exception to collaboration runtime environment
raiseException(currentException);

```

When the collaboration runtime environment executes this `raiseException()` call, it takes the following steps:

- Set the collaboration execution to the Exception state.
- Proceed with the execution path to determine how to terminate execution of the main diagram.
- Because the collaboration's execution is in the Exception state, create an unresolved flow when the collaboration ends. Obtain the exception message from the exception and associate it with the unresolved flow.
- Send the unresolved flow to the unresolved flow queue.

When the administrator views unresolved flows, the Flow Manager tools shows this unresolved flow's message (obtained from the exception when it first was thrown, down in subdiagramB).

Handling particular service-call exceptions

When a collaboration sends a business object request to its destination application, the collaboration runtime environment indicates any failure by throwing a collaboration exception with an exception type of `ServiceCallException`. However, the cause of a service-call failure can be ambiguous. A service call might fail for the following reasons:

- Application-related or logic-related—For example, a problem occurs because the entity that the collaboration attempted to retrieve does not exist in the application. In such a case, the application does not create or modify the requested entity.
- Transport-related—For example, a problem occurs during the transmission of the business object between the collaboration and the application. In this case, the application may have processed the request but transmission of the return status fails to reach the collaboration.

This section provides information about how to handle the following service-call error conditions in your collaboration template:

- "Service calls and exactly-once requests"
- "Unsent service call requests" on page 181

Service calls and exactly-once requests

The potential for duplication of data can occur in any of the following situations:

- If the service-call failure occurs during transmission of the business object from the application to the collaboration, and the collaboration object had completed or was in the process of completing one or more service calls, resubmission of the request during the recovery process could cause duplicate data.

- If InterChange Server ICS crashes while a collaboration object is processing a service call. In this case, the flow is considered to be In-Progress and the recovery process re-executes the entire scenario. For transactional collaborations, the failed service call is executed only after all compensation steps have been executed.

Preventing duplication of data due to transport-related exceptions must be handled at both of the following points:

- Collaboration runtime
- Boot-time recovery

The next sections describe how to handle the transport-related exceptions.

Handling runtime transport-related exceptions

To prevent duplication of data from a transport failure that occurs at collaboration runtime, code your collaboration template to distinguish after each service call between a transport-related exception and one that is not transport-related. To check for a transport-related exception, use the `ServiceCallTransportException` subtype of the `ServiceCallException` exception type. This exception subtype indicates that there was an error in the transport, and that it cannot be determined with certainty whether the request reached the application.

Important: A subset of exception types that previously were represented by the `AppUnknown` subtype of `ServiceCallException` are now represented by the `ServiceCallTransportException` subtype. Therefore, a collaboration template that checks specifically for the `AppUnknown` subtype must now also check for `ServiceCallTransportException`. Because the `AppUnknown` subtype no longer handles transport exceptions, the collaboration object will not trap transport exceptions if it checks for all subtypes except `ServiceCallTransportException`.

Handle a transport-related exception in the node immediately following the exception branch; that is, the node to which the exception branch points. Code the exception-handling node that catches an exception with the `ServiceCallException` exception subtype to provide an extra Retrieve service call that retrieves the request business object from the destination application and determines whether the application successfully created or modified the object. If the object was *not* created or modified successfully, code the collaboration to re-try the request.

The following code fragment provides exception handling for a transport-related exception. It uses the `getSubType()` method to retrieve the exception subtype from the `currentException` system variable. If this exception subtype is `ServiceCallTransportException`, the exception-handling code must perform a retrieve to determine whether data has changed in the application as a result of the service call request.

```
if (currentException.getType().equals(ServiceCallException)) {
    if (currentException.getSubType().equals(
        ServiceCallTransportException))
    {
        //Perform a retrieve to determine whether data changed
        //in the application reflecting the ICS business object request
    }
    else
        raiseException(ServiceCallException, ...);
}
```

Note: Checking for a transport-related exception is particularly important if the collaboration connects to the destination application over an unreliable network. In such a case, it is important to retry every time this exception could occur. Because the code can check specifically for exceptions caused by transport failure, there is a much lower performance impact than if your code performs the retry for *all* exceptions.

Handling boot-time recovery transport-related exceptions

To prevent duplication of data for a failure that results from a crash of InterChange Server while a collaboration service call was in process, you can do either of the following:

- Make the collaboration transactional, providing compensation for every service call.
- Configure a non-transactional collaboration for Service Call In-Transit persistence.

Transactional collaborations: When InterChange Server recovers a transactional collaboration that specifies compensation for every service call includes rolling back the collaboration before resubmitting the failed request. Because of the rollback, duplication of data for a failure resulting from a server crash is not a problem. For more information, see “Using transactional features” on page 143.

Non-transactional collaborations: Recovery of a non-transactional collaboration does *not* include rolling back the collaboration before resubmitting the failed request. Therefore, duplication of data can be a problem. To prevent data duplication for a non-transactional collaboration, configure the collaboration object for Service Call In-Transit persistence. On the Collaboration Properties dialog, check the box labelled:

Persist Service Call In Transit State

For backward compatibility with existing collaborations and because transactional collaborations do *not* benefit from persistent storage of each service call state, the default setting for Service Call In-Transit persistence is off; that is, the Persist Service Call In Transit State box is not checked for a collaboration object.

When a collaboration object that has been configured for persistence enters a service call, ICS maintains the state of the request until it has been completed. If the server crashes while a collaboration is processing a service call, the state of that failed service call is displayed in the Flow Manager, with the following status:

- An event status of “Service Call In Transit”
- A message with the following text:
Service Call In Transit

In this case, the administrator must manually determine whether the collaboration request was successfully processed before the transport failure. If the request was *not* successful, the administrator should resubmit it. For more information, see the *System Administration Guide*.

Note: If a collaboration is *not* configured for Service Call In-Transit persistence, all runtime failures and exceptions that cause the flow to fail have an event status of “Failed” in the Unresolved Flows viewer.

Unsent service call requests

To verify that the service call request has been sent to the application, code your collaboration template to check after each service call. To verify that the request has been sent, use the `AppRequestNotYetSent` subtype of the `ServiceCallException` exception type. In the case of a parallel connector agent, this exception subtype indicates that the request was queued up in the agent master but never got dispatched to the application; therefore, you can resend the request.

Handle an unsent service-call exception in the node immediately following the exception branch; that is, the node to which the exception branch points. Code the collaboration template to resubmit the request. The following code fragment provides exception handling for an unsent service call request. It uses the `getSubType()` method to retrieve the exception subtype from the `currentException` system variable. If this exception subtype is `AppRequestNotYetSent`, the code must resubmit the event by returning to the action node with the service call.

```
if (currentException.getType().equals(ServiceCallException))
{
    if (currentException.getSubType().equals(
        AppRequestNotYetSent))
    {
        // Resubmit the event by returning execution to the action node
        // with the service call.
    }
    else
        raiseException(ServiceCallException, ...);
}
```

Important: The collaboration runtime environment does *not* set a value for the `AppRequestNotYetSent` subtype if the `ControllerStoreAndForward` connector property of the destination connector is set to true. If this connector property is set to false, the collaboration should check for this subtype and resend the request.

Exceptions from the Collaboration API

When you design for a collaboration template, you can include decision nodes with exception branches to catch exceptions thrown by the methods of the Collaboration API. For those methods that throw a `CollaborationException` exception, the reference description has a section entitled `Exceptions`, which lists when exceptions are thrown by that method.

Chapter 8. Workspace and layout options

This chapter contains information on options for arranging symbols and customizing the workspace when you are editing an activity diagram.

For additional information about customizing the layout of the Process Designer main window, see “Customizing the main window” on page 23.

Aligning symbols

The alignment operations in the Alignment toolbar (see Figure 64) reposition two or more symbols to line up specified edges or centers. The Alignment toolbar becomes active when more than one symbol is selected in the activity diagram.

The order for all alignment operations is:

1. Select a symbol to use as the “base” (or anchor) for the alignment.
2. While holding down the Shift key, select the other symbols that you want to align to the first.
3. On the Alignment toolbar, click the operation you want to perform. See Figure 64.

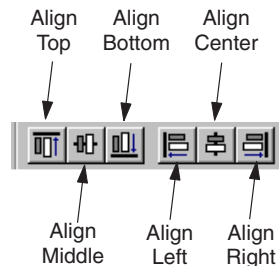


Figure 64. Alignment toolbar

Aligning edges

Aligning the edges of multiple symbols aligns the specified edge of each symbol to an imaginary line that runs along the specified edge of the model symbol. Edge-alignment operations include: Align Top, Align Bottom, Align Left, and Align Right.

For example, Figure 65 illustrates the result of aligning the bottoms of an End Success symbol and an Action symbol.



Figure 65. Aligning bottoms

For the End Success symbol, the label and the symbol together form one object whose bottom is aligned to the bottom of the Action symbol.

To align the top, bottom, left, or right edge of a set of symbols:

1. Click the symbol (base or anchor) to which you want to align the others.
2. While holding down the Shift key, click one or more additional symbols or groups of symbols.
3. On the Alignment tool bar, click the Align Top, Align Bottom, Align Left Sides, or Align Right Sides button.
All symbols line up to the target.

Aligning centers

You can center symbols along an imaginary horizontal or vertical line drawn at the center of the first symbol that you select. Each symbol is then centered horizontally or vertically along that line. Center-alignment operations include Align Middle and Align Center.

The dashed line in Figure 66 illustrates the Align Middle operation: the alignment of the vertical centers of two symbols.

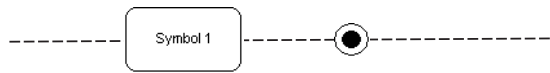


Figure 66. Align Middle operation

The dashed line in Figure 67 illustrates the Align Center operation: the alignment of the horizontal centers of two symbols.

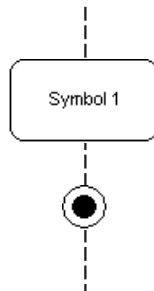


Figure 67. Align Center operation

To align centers:

1. Select the symbol or pregrouped set of symbols whose center you want to use as a base or anchor.
2. While holding down the Shift key, select the other symbols or groups of symbols that you want to align.
3. On the Alignment toolbar, click Align Middle or Align Center.

For example, these are two symbols before alignment of their horizontal centers:

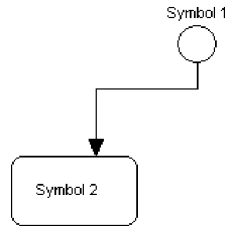


Figure 68. Unaligned symbols

These are the same symbols with their horizontal centers aligned:

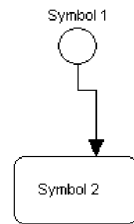


Figure 69. Aligned symbols

Nudging symbols

The “nudge” operations in the Nudge toolbar (see Figure 70) allow you to slightly move selected symbols of an activity diagram. The Nudge toolbar becomes active when a symbol is selected in the activity diagram.

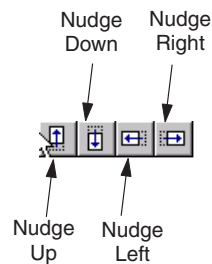


Figure 70. Nudge toolbar

To control finer movement of symbols:

1. Click the symbol that you want to move.
2. While holding down the Shift key, select the other symbols or groups of symbols that you want to move.
3. On the Nudge tool bar, click the Nudge Up, Nudge Down, Nudge Left, or Nudge Right button.

By default, these commands move the selected components in the specified direction by one pixel unit. You hold down the SHIFT key at the same time to move the selected components by five pixel units.

Zooming or panning on symbols

The operations in the Zoom/Pan toolbar (see Figure 71) allow you to zoom or pan selected symbols of an activity diagram. The Zoom/Pan toolbar becomes active when a symbol is selected in the activity diagram.

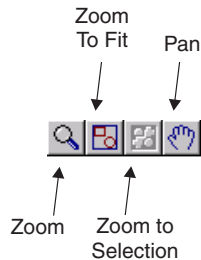


Figure 71. Zoom/Pan toolbar

The Zoom/Pan toolbar provides the following operations:

- **Zoom:** this operation allows you to zoom in on selected symbols in the diagram. To perform a zoom, click and hold the left mouse button while in zoom mode. When you hold the mouse button and drag in zoom mode, zoom-selection draws a rectangle to indicate the area into which you are zooming. After you have positioned in the area of the diagram, release the mouse button to select the area for zooming.
- **Zoom-to-Fit:** this operation sets the diagram magnification so that all symbols in the diagram are visible.
- **Zoom-to-Selection:** this operations lets you zoom on selected symbols. Select the symbol and click Zoom-to-Selection to magnifies the symbol to fit the frame.
- **Pan:** this operation allows you to “pan” or move to different areas of the diagram. After you click Pan, the pointer changes to a hand. You then click and drag the mouse to move around the diagram.

Using the workspace grid

Process Designer displays a grid in the workspace of the diagram editor to help you align symbols. In the Grid Properties dialog (see Figure 72), you can set the following grid options:

- Set the grid to be visible or invisible.
You can manually line up symbols to the grid if it is visible. By default, the grid is not visible.
- Turn on “snap to grid”.
- Set the grid color.
- Set grid spacing.

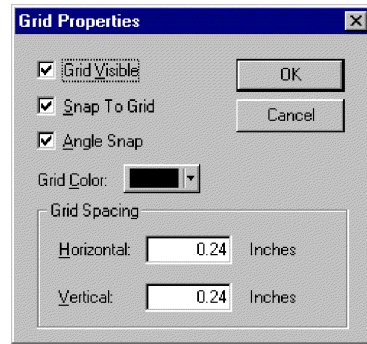


Figure 72. Grid Properties dialog

To change grid properties:

1. Ensure the diagram editor window is open.
2. Click View → Grid Properties to open the Grid Properties dialog box.
3. Adjust any of the following aspects of the grid, as needed:
 - Whether the grid is visible or invisible
 - Whether to set symbol movement so that it snaps to grid lines
 - The color of the grid
 - The grid spacing

You can adjust the size of the squares in the grid to your preferred measurements. You can adjust the grid size even if the grid is invisible.

Under Grid Spacing, enter a number for the width and height of each grid square, in inches.

The Angle Snap selection does not affect symbols in the activity diagram.

4. Click OK to save the grid options. Click Cancel to cancel any changes. Both options close the dialog.

You can also control the following grid options directly from the View menu:

- The Grid option controls the visibility of the grid.
- The Snap to Grid option controls whether symbols move to a grid line.

Changing display: user preferences

Process Designer provides several ways you can change its display. Access the User Preferences window by clicking View → Preferences, or use the shortcut key combination **Ctrl+U**.

The User Preferences window has three tabs that allow the following customizations:

- “Changing general display”
- “Changing diagram display” on page 188
- “Changing the color of symbols and links” on page 189

Changing general display

The General tab of the User Preferences window allows you to:

- Enable and disable the display of Workbook tabs at the bottom of Process Designer’s working area. Figure 4 on page 16 illustrates a working area that displays three Workbook tabs.

Note: Changing this display becomes effective only when you restart Process Designer.

- Enable and disable content validation. Disabling content validation is useful if a template has become corrupted but you want to inspect it anyway. However, disabling content validation does not guarantee that Process Designer can open a corrupted template.
- Enable and disable template content compression. This option is not user-configurable. It is provided to show the current compression setting. Compressed content can enhance the client-server data transfer rate and storage requirement.
- Display or hide the standard imports section and Ports and Triggering Events section in the Template Definition Declarations tab.
- Set the colors for each section of the Template Definitions Declarations tab and the Scenario Variables window of the Scenario Definitions dialog box.

Figure 73 illustrates the General tab of the User Preferences window.

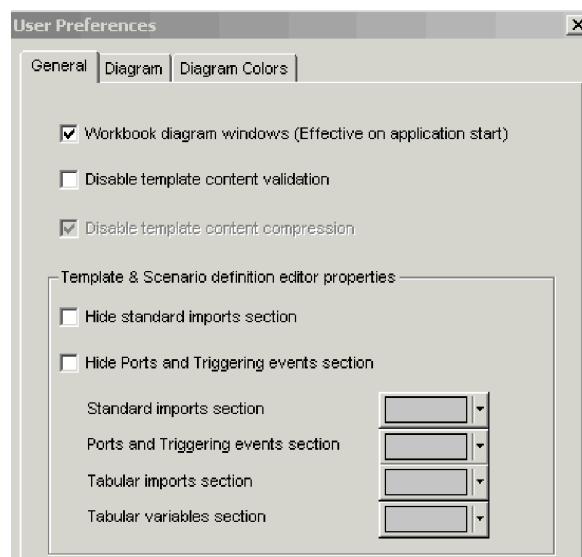


Figure 73. User Preferences dialog box: General tab

Changing diagram display

The Diagram tab of the User Preferences window allows you to:

- Enable and disable automatic display of a start node in a blank new diagram. Disabling this display requires the developer to manually add the start node in each diagram.
- Enable and disable the display of connection points on nodes in an activity diagram. Displaying connection points facilitates adding a transition link between two nodes.
- Enable and disable the display of unused branches in a decision node.
- Enable in-place editing of code fragments in the Action Properties dialog box. If these options are enabled, you can edit code fragments directly in the Action Properties dialog box instead of using Activity Editor.

Figure 74 illustrates the Diagram tab of the User Preferences window.

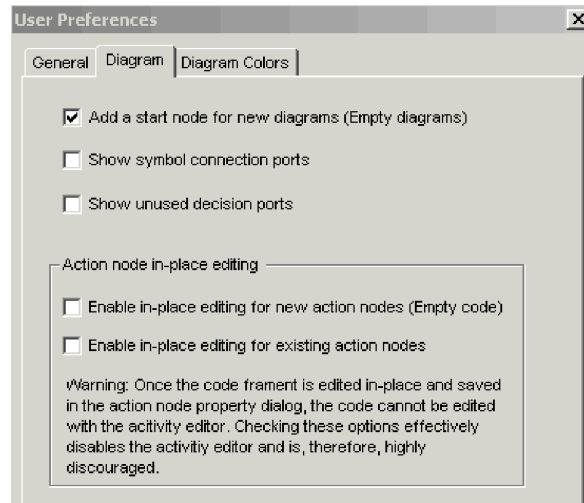


Figure 74. User Preferences dialog box: Diagram tab

Changing the color of symbols and links

The Diagram Colors tab of the User Preferences window allows you to change the display color of symbols and links in an activity diagram. For each symbol or link, press the arrow for the fill or line color field and make your selection from the drop-down options. Figure 72 illustrates the Diagram Colors preferences window.

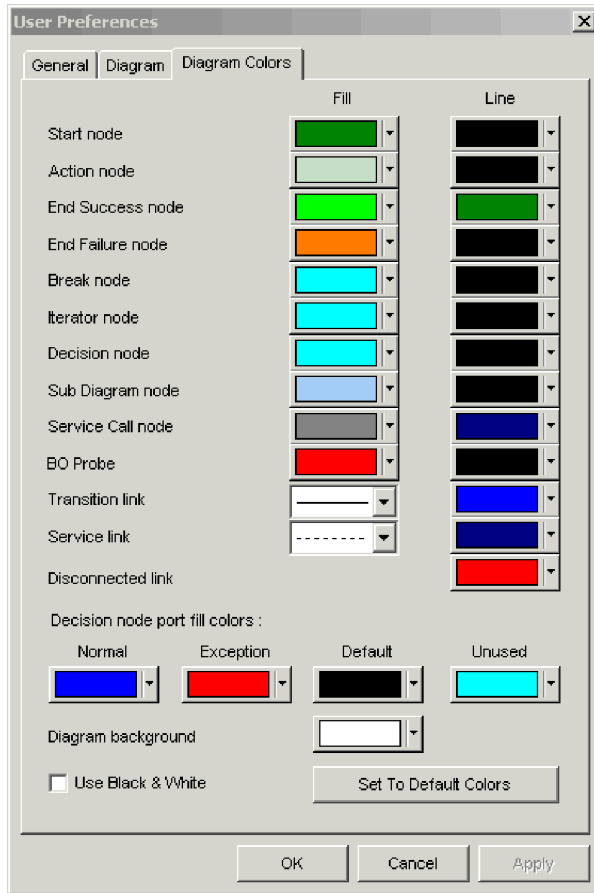


Figure 75. User Preferences dialog box: Diagram Colors tab

Aligning scenarios

There may be times during the development of a template that a scenario may become so out of line that you would spend a significant amount of time realigning the scenarios. (See Figure 76.) As you can see, the various decision nodes, transition links and other objects are severely misaligned in this diagram. To correct the alignment would take a significant amount of manual work.

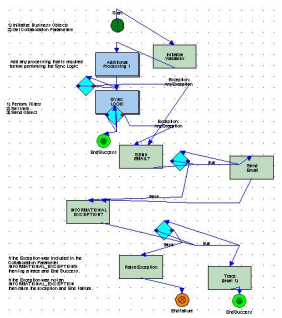


Figure 76. A misaligned scenario

To align a scenario, use the Auto Layout option of the View menu (see Figure 77). Selecting this option aligns the objects in the scenario so it displays correctly in the Process Designer (see Figure 78).

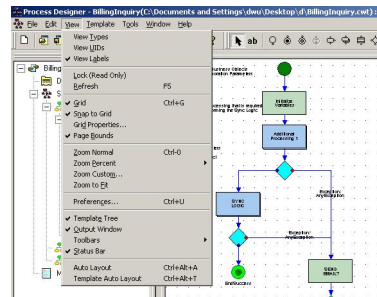


Figure 77. View menu

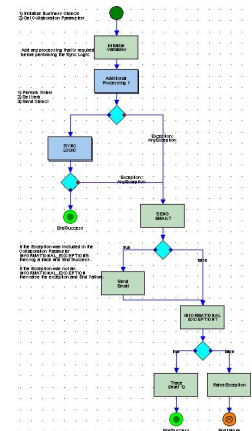


Figure 78. Scenario after Auto Layout is applied.

To align all of the scenarios inside a template at the same time, select the Template Auto Layout option of the View menu.

Hiding the Symbol Properties dialog boxes

When you double-click a symbol, its properties appear in a small dialog. When you click another symbol, the contents of the dialog display the newly-selected symbol's properties.

If you prefer not to see the Symbol Properties window, close it.

Chapter 9. Coding tips and examples

This chapter describes how to program specific types of collaboration tasks.

Operations on the collaboration

This section describes operations that affect the collaboration as a whole. It includes these operations:

- “Logging messages”
- “Adding trace messages” on page 195
- “Retrieving a collaboration configuration property” on page 197
- “Reusing collaboration object instances” on page 198
- “Calling a native map” on page 199

Each collaboration template must have an associated message file. The *message file* contains the text for the collaboration’s exception and logging messages. A unique number identifies each message in the message file. The text of the message may also include placeholder variables.

When the collaboration calls a method that displays a particular message, it passes the method the message’s identifying number and potentially additional parameters. The method uses the identifying number to locate the correct message in the message file, and it inserts the values of the additional parameters into the message text’s placeholder variables.

For example, a collaboration’s message file might include a message identified as number 23, whose text includes two placeholder variables, marked as {1} and {2}:

```
23  
Customer ID {1} could not be changed: {2}
```

When the collaboration wants to display or log this message, it passes the appropriate method, such as `raiseException()`, the identifying number of the message (23) and two additional parameters, the customer ID number (6701) and a String variable containing some additional explanatory text, such as greater than maximum length. The method locates the correct message, substitutes the parameter values for the message’s placeholders, and displays or logs the following message:

```
Customer ID 6701 could not be changed: greater than maximum length
```

Logging messages

A collaboration template can log a message whenever something occurs that might be of interest to an administrator. To log a message, use the `logInfo()`, `logWarning()`, and `logError()` methods in your collaboration template. Each method is associated with a different message severity level. Table 52 lists the severity levels and their associated methods.

Table 52. Message levels for log methods

Method	Severity level	Description
<code>logInfo()</code>	Info	Informational only. The user does not need to take action.
<code>logWarning()</code>	Warning	Represents information about a problem. Do not use this level for problems that the user must resolve.

Table 52. Message levels for log methods (continued)

Method	Severity level	Description
logError()	Error	Indicates a serious problem that the user needs to investigate.

The message text that these methods send to the log destination is prefixed with the severity level.

This section provides the following information about logging messages:

- “Using a message file”
- “Principles of good message logging”

Using a message file

Every collaboration template must have a message file to hold its log messages. When a collaboration logs an error, the text of the error message can come from the collaboration’s message file. The following example logs an error message whose text is contained in the collaboration’s message file:

```
logError(10, customer.get("LName"), customer.get("FName"));
```

The text of error message 10 has two message parameters and appears as follows in the message file:

```
10
Credit report error for {1} {2}.
```

When the logError() method executes, it obtains the text for message 10 from the message file, substituting the customer’s last name and first name substituted for message parameters 1 and 2, and prefixing the message with a severity of “Error”. It then writes this error message to the collaboration’s log destination.

For example, the logged message for a customer named John Davidson looks like this:

```
Error: Credit report error for Davidson John.
```

If the collaboration is configured for email notification, logError() also sends this error message to the designated email recipient (or recipients). Refer to Chapter 10, “Creating a message file,” on page 231, for information on how to set up a message file.

Principles of good message logging

When creating messages, keep in mind that the way that administrators use the logging feature.

Assigning severity levels: It is important to be precise when assigning error levels to messages. The InterChange Server email notification feature sends a message to a designated person, usually the administrator, when it detects the generation of an error message or fatal error message (logError()). Administrators use this InterChange Server email notification feature, and they additionally might link it to an email pager to send a page when an error occurs. By being precise when assigning error levels to messages, you can reduce the number of critical messages.

Revising messages: You can revise the text of a message at any time, such as to clarify or expand the text. However, after you assign a message number to a certain type of error, it is important that you do *not* reassign the number. Many administrators depend on scripts to filter log messages, and these scripts rely on

the message numbers. Thus, it is important that the numbers in the message file do not change meaning. If they do, users can lose messages or receive inadvertent messages.

When to use informational messages: You can use the `logInfo()` method to create temporary messages for your own debugging. However, be sure to remove these debugging method calls when you are finished with development.

Resist the temptation to use the `logInfo()` method to document the normal operation of the collaboration. Doing so fills the administrator's log files with messages that are not of interest. Instead, use the `trace()` method to give the administrator detailed information for debugging.

Adding trace messages

You can add trace messages to your collaboration template so that when a collaboration object runs, it generates a detailed description of its actions. Trace messages are useful for your own debugging and for on-site troubleshooting by administrators.

Trace messages differ from log messages in that trace messages are suppressed by default, whereas log messages cannot be suppressed. Trace messages are generally more detailed and are meant to be viewed only under certain circumstances, such as when someone intentionally configures the collaboration object's trace level to a number higher than zero. You can send trace messages and log messages to different files.

There are two types of trace messages for a collaboration:

- Collaboration-generated trace messages, which you code into the collaboration template
- WebSphere InterChange Server-generated or system-generated tracing messages from the collaboration runtime environment

Use the Collaboration Object Properties dialog box in System Manager to set the trace levels for both types of trace messages.

The collaboration template developer creates the levels for which you can request collaboration-generated tracing, as the next section describes. System-generated tracing levels are the same for all collaboration objects. They are described under "WebSphere InterChange Server-generated trace messages" on page 197.

Collaboration-generated trace messages

You can add trace messages to a collaboration template to report operations that are specific to that collaboration. Below are some examples of information that the collaboration can write to the trace file:

- Key values of a business object at the point that the collaboration enters or exits a particular action node.
- The value of a configuration option when the collaboration retrieves it.
- The decision to take a particular branch in the execution path.
- The exception code resulting from a service call.
- The value of each attribute in a business object at the point that it enters or exits a particular action node, iterator, or subdiagram.

Assigning trace levels: Each trace message must be associated with a **trace level** between 1 and 5. The trace level usually correlates to a level of detail: messages at

level 1 typically contain less detail than messages at level 2, which contain less detail than those at level 3, and so forth. Thus, if you turn on tracing at level 1, you see messages that contain less detail than the messages at level 5. However, you can assign levels in any way that is useful to you. Here are some suggestions:

- You can assign the same level to all of your trace messages.
- You can assign trace levels according to level of detail, as the collaboration runtime environment does.
- You can assign message levels according to the business object involved: level 1 traces messages relating to a certain business object, level 2 traces messages relating to another business object, and so on.

When you turn on tracing at a particular level, the messages associated with the specified level and those associated with all lower levels appear. For example, tracing at level 2 displays messages associated with both level 2 and level 1.

Tip: Make sure to note the tracing levels with your documentation, so users know what level to use when they need to trace.

Generating a trace message: The following is an example of a message and the method call that generates the message. The message appears in the message file as follows:

```
20  
Configuration property DO_VERIFICATION = {1}
```

The method call obtains the value of the configuration property DO_VERIFICATION, then uses the value to replace the parameter in the message. The code appears in the collaboration as follows, and the message appears when the user sets tracing to level 3:

```
String validateProp = getConfigProperty("DO_VERIFICATION");  
trace(3, 20, validateProp);
```

The following example obtains the value of an Employee business object's Salary attribute and makes a branching decision based on the amount of the salary. The message in the message file is:

```
15  
Salary {1} {2}
```

The example sends a trace message documenting the salary amount and the path taken.

```
int newsalary = employee.getInt("Salary");  
String sal = Integer.toString(newsalary);  
if (newsalary <150000)  
{  
    trace (3, 15, sal, "do extra check");  
}  
else  
{  
    trace (3, 15, sal, "take normal path");  
}
```

WebSphere InterChange Server-generated trace messages

The WebSphere InterChange Server collaboration runtime environment has a tracing component that provides messages about its execution of a collaboration.

The runtime environment tracing component uses six numbers to represent trace levels. The first level, zero, is the default setting, and it indicates that no tracing is occurring. Levels 1 through 5 each indicate an increasing level of detail. A level 1 trace provides the least detail and a level 5 trace provides the most detail.

To turn on tracing, change the collaboration object's trace level from zero to a higher number. Individual trace messages are associated with each level. Table 53 describes the types of messages that appear at each level.

Table 53. Trace levels for system-generated tracing

Level	WebSphere InterChange Server system-generated tracing
0	None.
1	The receipt of a triggering event and the start of a scenario.
2	The start and completion of a scenario, reporting both forward execution and rollback.
3	The execution of an action node.
4	The sending of a business object in a service call and the receipt of a response.
5	A detailed version of the trace at Level 4. This level trace prints the value of each attribute in the business object sent and received.

Retrieving a collaboration configuration property

To retrieve the collaboration's configuration property, use the `getConfigProperty()` method.

The following example shows how a collaboration can use a configuration property to determine the code path.

```
if (getConfigProperty("CONVERT_NEGQTY").equals("true")) {  
    // take this code path  
}  
else {  
    // take this code path  
}
```

To compare a property value with a specific value, always use an `equals()` method, as shown in the example. Do not use the conditional equality operator `==`, which tests whether two variables refer to the same object, rather than that two objects contain the same values. (See Table 37 and Table 38 for a complete list of available operators.)

Note that the value is case-sensitive. The case of the configuration parameter must be the same as the case in the code that tests for equality. In the preceding example, a value of `True` would fail the comparison.

A configuration property can also take an array of values, separated by semicolons. For more information, refer to "`getConfigPropertyArray()`" on page 346.

Reusing collaboration object instances

Typically, WebSphere InterChange Server creates an instance of a collaboration object to process each triggering event. When the instance completes the handling of the triggering event, the system frees up its resources and returns them to the Java free pool. However, the JDK does not always clean up these instances efficiently, which can lead to excessive memory usage.

To reduce memory usage, WebSphere InterChange Server uses the Collaboration Instance Reuse option, which allows the system to recycle an instance of a collaboration object by caching it and reusing it when the same type of collaboration object is instantiated at some later time. When WebSphere InterChange Server can recycle an existing collaboration instance, it can avoid:

- The overhead of collaboration-object instantiation
- Reliance on the JDK garbage collector for memory management

The system automatically uses the Collaboration Instance Reuse option as long as the collaboration template meets *both* of the following requirements:

- The collaboration does *not* contain any template (global) variables.
- The collaboration has been compiled with a version of Process Designer *after* version 3.0.

If either of these conditions is *not* met, the Collaboration Instance Reuse option is not used. Therefore, to take advantage of this option, avoid use of template (global) variables in the collaboration-template code. A template variable is a variable you declare whose scope is the entire collaboration template. You declare a template variable in the area labelled “Global Variables:” in the Declarations tab of the Definitions window.

If your collaboration requires template variables and you still wish to use the Collaboration Instance Reuse option, ensure that the collaboration template meets the following programming requirements:

- Avoid initializing template variables at declaration time. Instead, ensure that any template variables are always initialized in the first node of the collaboration template.
- If the collaboration template uses a template variable to reference a large object, make sure that you reset this variable to null before:
 - the collaboration exits
 - an exception is thrown

Important

A collaboration template containing template variables that are *not* initialized at the first node cannot safely be recycled because the variable values in the cached collaboration object instance persist when the instance is reused. When the cached collaboration instance is reused and begins execution, each template variable contains the value from the end of the previous use of the collaboration instance.

After you have coded your collaboration template to correctly initialize its template variables, perform the following tasks to enable the Collaboration Instance Reuse option:

1. Define a collaboration-specific configuration property called `EnableInstanceReuse` and set its default value to either `true` or `false`.
You define collaboration-specific configuration properties in the Template Definition window. Set the default value of `EnableInstanceReuse` according to the desired behavior of the collaboration objects:
 - To force instance recycling of *all* collaboration objects of the collaboration template, set the default value of `EnableInstanceReuse` to `true`.
 - To force instance recycling of only particular collaboration objects of the collaboration template, set the default value of `EnableInstanceReuse` to `false`.
2. Make sure that each collaboration object that is to be recycled has its `EnableInstanceReuse` collaboration property set to `true`.
You set the values of collaboration-specific configuration properties in the Properties tab of the Collaboration Object Properties window of System Manager. For more information, see the *System Administration Guide*.

If you cannot code your collaboration so that it meets the preceding programming requirements, do *not* use the Collaboration Instance Reuse option. To disable this option, do not define the `EnableInstanceReuse` collaboration configuration property for the collaboration template.

Note: You must stop and start the collaboration object to activate the Collaboration Instance Reuse option. It takes effect only after a subsequent reactivation of the collaboration.

The software uses a cache, called the *collaboration-instance cache*, to hold instances of collaboration objects. It derives the size of the collaboration-instance cache from the value of the “Maximum number of concurrent events,” which you configure in the General tab of the Collaboration Object Properties window of System Manager. You might need to resize the collaboration-instance cache depending on whether you are using the event-triggered or call-triggered flow-processing model to execute collaborations.

Resizing the collaboration-instance cache involves defining a collaboration configuration property called `CollaborationInstanceCacheSize`. You define this property with other collaboration properties, in the area labelled “Properties” in the General tab of the Template Definition window. After you define `CollaborationInstanceCacheSize`, set its value to a reasonable default value for the number of collaboration instances. For more information, see the description of the Collaboration Instance Reuse option in the *System Administration Guide*.

Calling a native map

Normally, calling maps and submaps is done only from within a map. However, sometimes your collaboration might need to call a WebSphere InterChange Server native map directly. Calling a native map from a collaboration provides a number of benefits:

- Easy generic-to-generic transformations
- Convenient translation between different application structures

Calling the native map from a collaboration template involves the following steps:

- “Creating a collaboration property for the map name” on page 200
- “Initializing the collaboration” on page 200
- “Calling the map” on page 200

- “Populating the collaboration variable” on page 203

Creating a collaboration property for the map name

You can create a collaboration property that contains the name of the map to call. This step is not required but it frees the collaboration code from needing to be recompiled if the map name changes. Instead, if the map name changes, you only have to change the value of this collaboration property. For example, you could define a collaboration property called `MAP_NAME` to hold the name of the map you need to call. Collaboration properties are defined in the Template Definitions window. For more information, see “Defining collaboration configuration properties (the Properties tab)” on page 91.

Note: Use the `getConfigProperty()` method to obtain the value of this collaboration property within the collaboration template’s code.

Initializing the collaboration

Initializing the collaboration template to call the map involves importing Java classes of the Mapping API into the collaboration template. WebSphere InterChange Server maps require certain Java classes to execute. Some of these classes are *not* automatically included in a collaboration template. For the map to be able to execute, you must explicitly import the following map class and packages:

- `CxCommon.CxExecutionContext` class
- `CxCommon.Exceptions` package
- `CxCommon.Dtp` package
- `CxCommon.BaseRunTimes` package
- `DLM` package

Import each of these items in the Imports section of the Declarations tab in the Template Definitions window. For example, add the following entries to the import table to import map classes into the collaboration template:

```
CxCommon.CxExecutionContext
CxCommon.Exceptions.*
CxCommon.Dtp.*
CxCommon.BaseRunTimes.*
DLM.*
```

Note: Make sure you follow the package name with the “.*” syntax to import *all* of the classes within the package.

For general information about how to import Java classes, see “Importing Java packages” on page 87.

Calling the map

To call a map, use the `runMap()` method of the Mapping API class, `DtpMapService`. The `runMap()` method requires the following information to be passed in as arguments:

- The name of the map to execute
- The type of map, always represented as `CWMAPTYPE` for WebSphere InterChange Server native maps
- An input array of business objects, which contains the source business objects for the map
- A map execution context

Therefore, you must initialize this information within the collaboration *before* the call to `runMap()`.

Obtaining the map name: To pass in the name of the map to execute, you can hardcode the map name in the call to `runMap()`. However, a more flexible design is to use a collaboration property to contain the name of the map. This design involves the following steps:

- Set a collaboration property, as described in “Creating a collaboration property for the map name” on page 200
- Use the `getConfigProperty()` method to obtain the value of this collaboration property within the collaboration template’s code.

Figure 79 shows a line of code obtains the map name stored in the collaboration property `MAP_NAME`.

```
String map_name = getConfigProperty("MAP_NAME");
```

Figure 79. Obtaining the Name of the Map to Execute

Note: This approach assumes that you have created the `MAP_NAME` collaboration property and assigned to it the correct name of the map to execute.

Initializing the input array: The `runMap()` method requires an input array, which contains the source business objects for the map. Usually, a map transforms a single source business object. For such a map, this input array has only one element. When calling a map from within a collaboration, you usually want to put a copy of the triggering business object into this input array. You then pass this input array as the third argument to `runMap()`.

Figure 80 shows a line of code that initializes the input array with a copy of the collaboration’s triggering business object.

```
BusObj[] sourceBusObjs = { inputBusObj };
```

Figure 80. Initializing the Input Array with the Triggering Business Object

Preparing the map execution context: A map instances executes within a specific map execution context, which contains information that the map needs, such as the following:

- The calling context indicates the condition that initiated this invocation of the map. Calling contexts are represented as predefined constants in the `MapExeContext` class.
- The original-request business object is a copy of the business object associated with this invocation of the map.

The Mapping API represents a map execution context as a `MapExeContext` object. Within map code, you can always obtain the map’s execution context from the system-generated variable, `cxExecCtx`. However, no such system-generated variable is accessible from within the collaboration template. Instead, your collaboration must take the following steps:

- Instantiate a `MapExeContext` object with the `MapExeContext()` constructor.
- Assign the `MapExeContext` object to the global execution context.

The `CxExecutionContext` class represents the global execution context for the collaboration. Therefore, to initialize the map execution context, you must take the following steps:

- Instantiate a `CxExecutionContext` object with the `CxExecutionContext()` constructor.
- Assign the `MapExeContext` object to the `CxExecutionContext` object with the `setContext()` method.
- Provide the `MapExeContext` object with its calling context.
The `setInitiator()` method of the `MapExeContext` class sets the calling context (its deprecated term is “map initiator”). For more information on map execution contexts and the methods of the `MapExeContext` class, see the *Map Development Guide*.

Figure 81 shows a code fragment that initializes the map execution context with a calling context of `EVENT_DELIVERY` (converting from an application-specific business object to a generic business object) and an original-request business object of the triggering business object.

```
// Instantiate objects for the map execution context and the global
// execution context
map_exe_context = new MapExeContext();
global_exe_context = new CxExecutionContext();

// Assign the map execution context to the global execution context
global_exe_context.setContext(
    CxExecutionContext.MAPCONTEXT,
    map_exe_context);

// Initialize the map execution context
map_exe_context.setInitiator(MapExeContext.EVENT_DELIVERY);
```

Figure 81. Initializing the map execution context

Calling the `runMap()` method: After the collaboration template has initialized the map information, it can call the map. Calling the `runMap()` method involves two steps:

- Invoke the `runMap()` method.
This method is a static method within the Mapping API class, `DtpMapService`. Therefore, you do not need to instantiate a `DtpMapService` instance.
- Provide an output array for the `runMap()` return values.
In addition to the input array, the `runMap()` method also requires an output array, which `runMap()` populates with the map’s destination business objects and returns to the calling code. Usually, a map generates a single destination business object. For such a map, this output array has only one element.

Figure 82 shows the call to `runMap()` to execute the map with the following characteristics:

- Map name is identified by the `MAP_NAME` collaboration property (Figure 79).
- Map’s source destination business object is the collaboration’s triggering business object (Figure 80).
- Map’s execution context is initialized to a calling context of `EVENT_DELIVERY` and the triggering business object as the original-request business object (Figure 81).

```

BusObj[] destinationBusObjs = DtpMapService.runMap(
    map_name,
    CWMATYPE,
    sourceBusObjs,
    global_exe_context);

```

Figure 82. Calling `runMap()` to execute the map

Populating the collaboration variable

The map's destination business objects are available in the output array that `runMap()` returns. To send a destination business object out of the collaboration, you must copy it from the output array and into the appropriate collaboration variable. This collaboration variable is usually associated with the collaboration's To port. Therefore, the destination business object is usually copied to the `ToBusObj` collaboration variable.

Figure 83 uses the `BusObj.copy()` method to copy the single destination business object returned by the `runMap()` call in Figure 82 to the `ToBusObj` collaboration variable.

```
ToBusObj.copy(destinationBusObjs[0]);
```

Figure 83. Populating the collaboration variable

Operations on business objects

This section describes operations that involve manipulating business objects and their values. It includes these operations:

- "Creating a new business object"
- "Creating a child business object in a new business object" on page 204
- "Copying the triggering event" on page 205
- "Copying or duplicating a business object" on page 206
- "Using attribute values" on page 207
- "Setting attribute values" on page 209
- "Setting an attribute value to null" on page 212

Creating a new business object

Use the constructor method `new` to create a new business object.

Syntax

```
new BusObj(String busObjType)
```

Parameters

busObjType The name of a business object definition

Return value

An object of type `BusObj`.

Exceptions

`ObjectException` – Raised if the business object argument is invalid.

Notes

This method creates a business object with no values. You set the values when you populate the attributes.

If an attribute in the new business object is defined as a child business object or child business object array, you must explicitly create the child business object and associate it with the attribute; for more information, refer to “Creating a child business object in a new business object” on page 204.

Example

The following example uses the Customer business object definition to create a new business object called `destinationCustomer`. The new business object is created but has no attribute values.

```
BusObj destinationCustomer = new BusObj("Customer");
```

Creating a child business object in a new business object

When you create a new business object, an attribute that is defined to contain a child business object of cardinality one or n has no value. For that attribute to have value, you must explicitly create a child business object or child business object array and then associate it with the attribute. This section shows how to do that for a single child business object and for an array.

Creating a single child business object

The following steps illustrate first creating a new business object and then creating a child business object that is contained in one of its attributes:

1. Use the new method to create a parent business object.
2. Use the new method to create one child business object of the type for which the attribute is defined.
3. Use the `BusObj.set()` method to set the attribute value in the parent object to the new child business object.

The following example illustrates the creation of a new Invoice business object, which has an attribute called `SoldToAddressAttribute`. This attribute holds the address of the sold-to customer and is a business object of type `Address`. The example shows the association between the parent business object and the child business object.

```
// Declarations
BusObj invoice = new BusObj("Invoice");
// Create child business object in invoice
invoice.set("SoldToAddressAttribute", new BusObj("Address"));
```

If the collaboration needs to manipulate the child business object, the example might look like this:

```
// Declarations
BusObj invoice = new BusObj("Invoice");
BusObj soldToAddress = new BusObj("Address");
// Manipulate child business object soldToAddress
// Associate child business object soldToAddress with parent invoice
invoice.set("SoldToAddressAttribute", soldToAddress);
```

Creating a child business object array

In this section, the following steps illustrate creating a new business object and then creating a child business object array that is contained in one of its attributes:

1. Use the new method to create a business object. This is the parent business object.
2. For the attribute in the parent that is defined to contain a business object with cardinality equal to n, create one business object of the attribute's specified type.
3. Set the parent's attribute value to the new single business object.
4. Declare a BusObjArray object, get the value of the attribute, and assign it to the array.

You can then use methods of the BusObjArray class to add elements or perform other operations on the business object array.

The following example illustrates the creation of a new Bill of Materials business object, the creation of a child business object array for its LineItems attribute, and the placing of additional business objects on the array.

```
// Declarations
BusObj bom = new BusObj("Bill_Of_Materials");
BusObjArray lineItemArray = null;
BusObj singleLineItem = new BusObj ("LineItem");
// Create first child item
bom.set("LineItemsAttribute", singleLineItem);
//If there are additional line items, do this once
lineItemArray = bom.getBusObjArray("LineItemAttribute");
// Now do this for each additional child item
lineItemArray.addElement(new BusObj("singleLineItem"));
```

Copying the triggering event

The first action of every scenario should handle the scenario's flow trigger. Process Designer automatically declares a variable of BusObj type called triggeringBusObj; this variable holds the flow trigger (triggering event or triggering access call) that caused the scenario to execute.

Process Designer also automatically declares template BusObj variables for each defined port. The name of the BusObj variable matches the port name, with BusObj appended.

For example, suppose the triggering event for a scenario is Customer.Create. The port that receives the triggering event is called SourceCust. Process Designer automatically declares a variable named SourceCustBusObj, combining the port name with the BusObj suffix. In this case, Process Designer displays the following variable declaration:

```
BusObj SourceCustBusObj = new BusObj("Customer");
```

You can add the following code fragment to copy the triggering event to SourceCustBusObj.

```
SourceCustBusObj.copy(triggeringBusObj);
```

Many collaborations are triggered by multiple types of business objects. You must first determine the type of the business object before you can create a business object to hold the flow trigger. Use the `BusObj.getType()` method on the flow trigger to first ascertain its type, then create the appropriate type of business object, and finally copy the flow trigger to the newly created business object.

```
sourceBusObj = new BusObj(triggeringBusObj.getType());  
sourceBusObj.copy(triggeringBusObj);
```

Tip: It is good programming practice to first copy the `triggeringBusObj` variable to another variable before you do any operations on it.

Copying or duplicating a business object

There are two methods that move values from one business object into another business object: `copy()` and `duplicate()`. Both methods deal with the entire hierarchy of a business object, copying the parent business object and all of its children. Table 54 describes both methods.

Table 54. Comparison of the `copy()` and `duplicate()` methods

Method	Description
<code>copy()</code>	Sets all the attribute values of an existing business object to those of another business object.
<code>duplicate()</code>	Creates a new business object by cloning an existing business object. Reproduces both the attribute values and the verb. The return value for this method must be assigned to a variable.

The difference between the two methods is mainly the preexistence of the business object where you want to put the copied values.

Copying

Before you use the `copy()` method, there must be an existing business object to which you want to copy values. Use the `new` method to create the business object if it does not exist.

The following example copies the attribute values contained in a `Customer` business object received from the source application to a business object that will be sent to the destination application:

```
BusObj destination = new BusObj("Customer");  
destination.copy(sourceBusObj);
```

Note: The `copy()` method copies the entire business object, including all child business objects and child business object arrays. This method does *not* set a reference to the copied object. Instead, it clones all attributes; that is, it creates separate copies of the attributes.

Duplicating

In contrast to the `copy()` method, `duplicate()` creates a complete clone of an existing business object and returns it.

The following example clones the source business object and assigns it to a variable called `destination`:

```
BusObj destination = sourceBusObj.duplicate();
```

Using attribute values

Collaborations frequently retrieve the values of attributes contained in business objects that they have received.

If a collaboration needs to do something with an attribute value it has received, it should check to make sure that the attribute value is not null before using it (see “Checking for nulls” on page 207).

Checking for nulls

To check for a null attribute value, a collaboration calls `BusObj.isNull(attribute)`. A null value usually appears for one of the following reasons:

- The attribute was never set.
Upon creation of a business object, all attributes are null and they remain null until explicitly set. These include child business objects and child business object arrays.
- The attribute was explicitly set to null by the `BusObj.set()` method.
- During the mapping process, there was no value in the input business object to map to this attribute.

The following code sample sets a value in a billing business object, using data received in an Order business object. If the received order data is itself null or blank, then the code sets the attribute to “Unknown.”

```
//Check whether ProductLine is null or blank; if so, default it
if (order.isNull("ProductLine") || order.isBlank("ProductLine"))
{
    logInfo("Setting ProductLine to default Unknown");
    order.set("ProductLine", "Unknown");
}
//Now set Billing object's equivalent attribute to the same value
billing.set("ProductLine", order.get("ProductLine"));
```

The `isNull()` method can be used on all types of attributes, including those that contain child business objects or child business object arrays.

Comparing an attribute value with a known value

A collaboration can use the Java programming language `equals()` method to check the value of an attribute against a specific, expected value. The `equals()` method compares the expected value with the retrieved attribute value, as the following example shows. Call `equals()` on the known object and compare it to the unknown attribute.

In this example, Smith is the value expected in the LName attribute.

```
String name = "Smith";
boolean checkName=name.equals(CustBusObj.get("LName"));
```

Retrieving the attribute

This section covers types of operations for retrieving attributes. These operations are presented in order of increasing complexity:

- Retrieving an attribute value of basic type
- Retrieving an attribute in a child business object

Retrieving an attribute of a basic type: The `BusObj.get()` methods retrieve attribute values that are of basic types. Basic types for an attribute are the supported primitives, as Table 55 shows.

Table 55. Retrieving attributes of basic data types

Basic data type	Method to set attribute
boolean	<code>getBoolean()</code>
double	<code>getDouble()</code>
float	<code>getFloat()</code>
int	<code>getInt()</code>
long	<code>getLong()</code>
Object	<code>get()</code>
LongText	<code>getLongText()</code>
String	<code>getString()</code>

The following example retrieves the contents of the Credit-Limit attribute, which is an int.

```
int creditLimit = customer.getInt("Credit-Limit");
```

If you do not know the data type of the attribute, use the form of the `get()` method that retrieves a Java Object data type.

Note: The `get()` method returns a copy of the attribute. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to attribute in the source business object is *not* made to the value that `get()` returns. Each time this method is called, it returns a new copy (clone) of the attribute.

Retrieving an attribute from a child business object: If an attribute is a business object type, the cardinality defined in the business object definition specifies whether the attribute contains a single business object or an array. If the cardinality is:

- 1, the attribute contains a single business object. Assign the return of `getBusObj()` to a `BusObj` object.
- n, the attribute contains an array of business objects. Assign the return of `getBusObjArray()` to a `BusObjArray` object.

The following example retrieves the single-cardinality Address business object contained in the SoldToAddress attribute of a Bill of Materials business object.

```
BusObj addr = new BusObj("Address");
addr = bom.getBusObj("SoldToAddress");
```

The following example searches for sold-to addresses in the United States. The business object structure is:

- BusOrg is the top-level business object.
- BusOrg has an attribute called SoldToSite, which is a multiple-cardinality business object.
- SoldToSite has an attribute called SoldToAddress, which is a single-cardinality business object.
- SoldToAddress contains an attribute called CountryName.

```
//Look for sold-to address in the US
//Start with the busOrg business object
//Get the child business object array in the "SoldToSite" attribute
if (!busOrg.isNull("SoldToSite"))
{
    BusObjArray siteAddArray = busOrg.getBusObjArray("SoldToSite");
    //
    //String to compare with sold-to country name
    String countryName = "USA";
    //Get size of child business object array
    int count = siteAddArray.size();
    //
    //For each business object in the array get the SoldToAddress
    //attribute, which is a business object, and compare its
    //SoldToCountryName attribute to the string "USA"
    //
    for (int i = 0; i < count ; i ++ )
    {
        BusObj siteAddr = siteAddArray.elementAt( i );
        if (!siteAddr.isNull("SoldToAddress"))
        {
            BusObj soldToAddress =
                siteAddr.getBusObj("SoldToAddress");
            if (countryName.equalsIgnoreCase(
                soldToAddress.getString("SoldToCountryName")))
            {
                //do something
            }
        }
    }
}

} //end for
} //end if
```

Setting attribute values

This section covers three types of operations for setting attributes. These operations are presented in order of increasing complexity:

- Setting an attribute value of basic type
- Setting an attribute in a single child business object
- Setting an attribute in a child business object that is part of a business object array

Setting an attribute of a basic type

The `BusObj.set()` methods set attribute values that are of basic types. Basic types for an attribute are the supported primitives: `boolean`, `double`, `float`, `int`, `long`, `Object`, and `String`.

All the `set()` methods have the same name but their signatures differ: the first parameter is always a `String` containing the name of the attribute whose value is to be set. The second parameter is a variable or constant that is a primitive type, a `String` object, or an `Object` type. You call the same `set()` method regardless of the type of the variable; the method is overloaded so that it accepts a variable of any type. The compiler determines which variation of the method to use.

The following example sets the values of two attributes: `FirstName` and `Salary`.

```
Customer.set("FirstName", "Sue");
Customer.set("Salary", 30500);
```

You can use the `BusObj.get()` and `BusObj.set()` methods to copy attribute values from one business object to another. The following example gets the `String` variables `HeaderId` and `ServiceId` from the source business object and sets the attributes `HeaderId` and `SalesNum` in the destination business object to those values.

```
DestinationBusObj.set(
    "HeaderId", SourceBusObj.getString("HeaderId"));
DestinationBusObj.set(
    "SalesNum", SourceBusObj.getString("ServiceId"));
```

Note: The `set()` method sets an object reference to the value when it assigns the value to the attribute. It does *not* clone the attribute value from the source business object. Therefore, any changes to the value in the source business object are also made to the attribute in the business object that calls `set()`.

Setting an attribute in a single child business object

To set an attribute in a child business object whose cardinality is equal to 1, you must first get a handle to the child business object.

Use the `BusObj.getBusObj()` method to get a handle or reference to the child business object and assign the results to a `BusObj` type variable. Then call the `BusObj.set()` method; this invokes the overloaded version of the method that matches the data type of the attribute.

The example that follows is based on Figure 84. The `Customer` business object has an attribute called `Address`, which is a reference to a child business object called `CustAddress`.

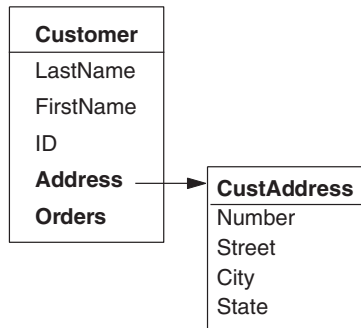


Figure 84. Single child business object

The code example does the following:

1. Declares a variable called addr
2. Retrieves the Customer business object's Address attribute and assigns it to addr
3. Sets the City attribute

```
BusObj addr = Customer.getBusObj("Address"); addr.set("City", "SF");
```

Setting an attribute in an array of child business objects

To set an attribute in an array of child business objects (an attribute whose cardinality is equal to n), use the `BusObj.getBusObjArray()` method and assign the results to a `BusObjArray` type variable. Then use the `BusObjArray` methods on that variable.

The code examples that follow are based on the following structure.

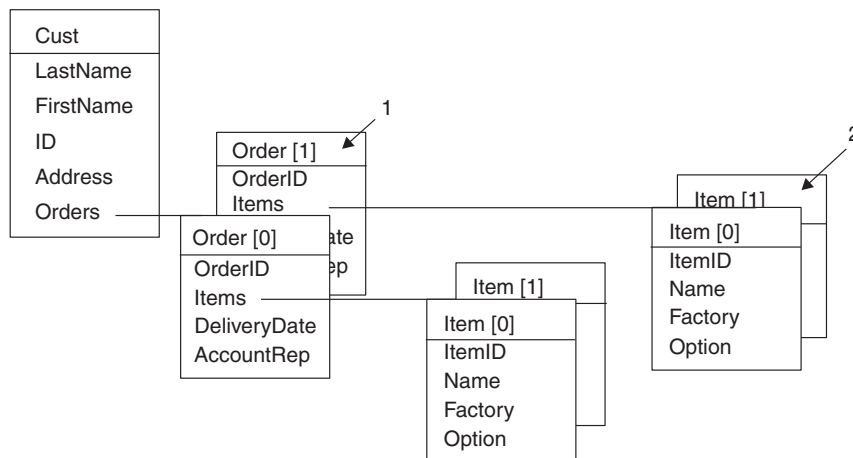


Figure 85. Child business object arrays

The following example sets the `OrderID` attribute of the business object marked 1 in Figure 85.

```
BusObjArray[] orders =
    cust.getBusObjArray("Orders").elementAt(1);
orders[1].set("OrderID", "x1234");
```

The following example sets the Factory attribute of the business object marked 2 in Figure 85.

```
BusObjArray items = orders[1].getBusObjArray("Items");
BusObj item = items.elementAt(1);
item.set("Factory", "MyCompany");
```

Setting an attribute value to null

The following example sets the value of the Total attribute of the order business object to null:

```
order.set("Total", null);
```

You can use this technique to set any type of attribute to null, whether the attribute value is a basic type, BusObj type, or BusObjArray type. However, you cannot use it to set child business objects in an array to null.

Executing database queries

During execution of a collaboration, you might need to obtain information from a database, such as the relationship database. To obtain or modify information from a database, you query its tables. A **query** is a request, usually in the form of an SQL (Structured Query Language) statement, that you send to the database for execution. Table 56 shows the steps involved in executing a query in a database.

Note: You can access any external database that WebSphere InterChange Server supports through JDBC through the Oracle thin type 4 driver and a WebSphere InterChange Server branded MS-SQL Server type 4 driver.

Table 56. Steps for executing a query

Task for executing a query	For more information
1. Obtain a connection (which is a <code>CwDBCConnection</code> object) to the database.	"Obtaining a connection" on page 213
2. Through the <code>CwDBCConnection</code> object, send queries and manage transactions in the database.	"Executing the query" on page 213 "Managing the transaction" on page 224
3. Release the connection.	"Releasing a connection" on page 228

Tip: One possible use of database queries is to handle service calls with long latency. After the collaboration issues a service request that is expected to take a long time, the collaboration saves the execution context using a database connection and then exits. The actual response from the service call, which might occur hours or even days later, returns as a new event and triggers another collaboration, which restores the proper execution context from the database and resumes the business process execution.

Obtaining a connection

To be able to query the database, you must first obtain a connection to this database with the `getDBConnection()` method of the `BaseCollaboration` class. To identify the connection to obtain, specify the name of the connection pool that contains this connection. All connections in a particular connection pool are to the same database. The number of connections in the connection pool is determined as part of the connection pool configuration. You must determine the name of the connection pool that contains connections for the database you want to query.

Important: Connections are opened when InterChange Server boots or dynamically, when a new connection pool is configured. Therefore, the connection pool that contains connections to the desired database must be configured before the execution of the collaboration object that requests the connection. You configure connection pools within System Manager. For more information, see the *Implementation Guide for WebSphere InterChange Server*.

In Figure 86, the call to `getDBConnection()` obtains a connection to the database that is associated with connections in the `CustDBConnPool` connection pool.

```
CwDBConnection connection = getDBConnection("CustDBConnPool");
```

Figure 86. Obtaining a connection from a connection pool

The `getDBConnection()` call returns a `CwDBConnection` object in the `connection` variable, which you can then use to access the database associated with the connection.

Tip: The `getDBConnection()` method provides an additional form that allows you to specify the transaction programming model for the connection. For more information, see “Managing the transaction” on page 224.

Executing the query

Table 57 shows the ways that you can execute SQL queries with methods of the `CwDBConnection` class.

Table 57. Executing SQL queries with `CwDBConnection` methods

Type of query	Description	<code>CwDBConnection</code> method
Static query	The SQL statement is sent as text to the database.	<code>executeSQL()</code>
Prepared query	After its initial execution, the SQL statement is saved in its compiled, executable form so that subsequent executions can use this precompiled form.	<code>executePreparedSQL()</code> on page 401
Stored procedure	A user-defined procedure that contains SQL statements and conditional logic	<code>executeSQL()</code> <code>executePreparedSQL()</code> on page 401 <code>executeStoredProcedure()</code>

Executing static queries

The `executeSQL()` method sends a static query to the database for execution. A **static query** is an SQL statement sent as a string to the database, which parses the string and executes the resulting SQL statement. This section covers how to send the following kinds of SQL queries to a database with `executeSQL()`:

- Queries that return data from the database (SELECT)
- Queries that modify data in the database (INSERT, UPDATE, DELETE)

- Queries that execute stored procedures defined in the database

Executing static queries that return data (SELECT): The SQL statement SELECT queries one or more tables for data. To send a SELECT statement to the database for execution, specify a string representation of the SELECT as an argument to the `executeSQL()` method. For example, the following call to `executeSQL()` sends a SELECT of one column value from the customer table:

```
connection.executeSQL(
    "select cust_id from customer where active_status = 1");
```

Note: In the preceding code, the `connection` variable is a `CwDBConnection` object obtained from a previous call to the `getDBConnection()` method (see Figure 86).

You can also send a SELECT statement that has parameters in it by using the second form of the `executeSQL()` method. For example, the following call to `executeSQL()` performs the same task as the previous example except that it passes the active status as a parameter to the SELECT statement:

```
Vector argValues = new Vector();

String active_stat = "1";
argValues.add( active_stat );
connection.executeSQL(
    "select cust_id from customer where active_status = ?", argValues);
```

The SELECT statement returns data from the database tables as rows. Each row is one row from the data that matches the conditions in the WHERE clause of the SELECT. Each row contains the values for the columns that the SELECT statement specified. You can visualize the returned data as a two-dimensional array of these rows and columns.

Tip: The syntax of the SELECT statement must be valid to the particular database you are accessing. Consult your database documentation for the exact syntax of the SELECT statement.

To access the returned data, follow these steps:

1. Obtain one row of data.
2. Obtain column values, one by one.

Table 58 shows the methods in the `CwDBConnection` class that provide access to the rows of returned query data.

Table 58. CwDBConnection methods for row access

Row-access task	CwDBConnection method
Check for existence of a row.	<code>hasMoreRows()</code>
Obtain one row of data.	<code>nextRow()</code>

Control the loop through the returned rows with the `hasMoreRows()` method. End the row loop when `hasMoreRows()` returns false. To obtain one row of data, use the `nextRow()` method. This method returns the selected column values as elements in a Java `Vector` object. You can then use the `Enumeration` class to access the column values individually. Both the `Vector` and `Enumeration` classes are in the `java.util` package.

Table 59 shows the Java methods for accessing the columns of a returned query row.

Table 59. Java methods for column-value access

Column-access task	Java method
Determine number of columns.	<code>Vector.size()</code>
Cast Vector to Enumeration.	<code>Vector.elements()</code>
Check for existence of a column.	<code>Enumeration.hasMoreElements()</code>
Obtain one column of data.	<code>Enumeration.nextElement()</code>

Control the loop through the column values with the `hasMoreElements()` method. End the column loop when `hasMoreElements()` returns false. To obtain one column value, use the `nextElement()` method.

The following code sample gets an instance of the `CwDBConnection` class, which is a connection to a database that stores customer information. It then executes a `SELECT` statement that returns only one row, which contains a single column, the company name "CrossWorlds" for the customer id of 20987:

```
CwDBConnection connectn = null;
Vector theRow = null;
Enumeration theRowEnum = null;
String theColumn1 = null;

try
{
    // Obtain a connection to the database
    connectn = getDBConnection("sampleConnectionPoolName");
}

catch(CwDBConnectionFactoryException e)
{
    System.out.println(e.getMessage());
    throw e;
}

// Test for a resulting single-column, single-row, result set
try {
    // Send the SELECT statement to the database
    connectn.executeSQL(
        "select company_name from customer where cust_id = 20987");

    // Loop through each row
    while(connectn.hasMoreRows())
    {
        // Obtain one row
        theRow = connectn.nextRow();
        int length = 0;
        if ((length = theRow.size())!= 1)
        {
            return methodName + "Expected result set size = 1," +
                " Actual result state size = " + length;
        }

        // Get column values as an Enumeration object
        theRowEnum = theRow.elements();

        // Verify that column values exist
        if (theRowEnum.hasMoreElements())
        {
            // Get the column value
            theColumn1 = (String)theRowEnum.nextElement();
            if (theColumn1.equals("CrossWorlds")==false)

```

```

        {
            return "Expected result = CrossWorlds,"
                + " Resulting result = " + theColumn1;
        }
    }
}

// Handle any exceptions thrown by executeSQL()
catch(CwDBSQLException e)
{
    System.out.println(e.getMessage());
}

```

The following example shows a code fragment for a SELECT statement that returns multiple rows, each row containing two columns, the customer id and the associated company name:

```

CwDBConnection connectn = null;
Vector theRow = null;
Enumeration theRowEnum = null;
Integer theColumn1 = 0;
String theColumn2 = null;

try
{
    // Obtain a connection to the database
    connectn = getDBConnection("sampleConnectionPoolName");
}

catch(CwDBConnectionFactoryException e)
{
    System.out.println(e.getMessage());
    throw e;
}

// Code fragment for multiple-row, multiple-column result set.
// Get all rows with the specified columns, where the
// specified condition is satisfied
try
{
    connectn.executeSQL(
"select cust_id, company_name from customer where active_status = 1");

    // Loop through each row
    while(connectn.hasMoreRows())
    {
        // Obtain one row
        theRow = connectn.nextRow();

        // Obtain column values as an Enumeration object
        theRowEnum = theRow.elements();
        int length = 0;
        if ((length = theRowEnum.size()) != 2)
        {
            return "Expected result set size = 2," +
                " Actual result state size = " + length;
        }
        // Verify that column values exist
        if (theRowEnum.hasMoreElements())
        {
            // Get the column values
            theColumn1 =
                ((Integer)theRowEnum.nextElement()).intValue();
            theColumn2 = (String)theRowEnum.nextElement();
        }
    }
}

```



```

    }
    catch(CwDBSQLException e)
    {
        System.out.println(e.getMessage());
    }
}

```

Note: The SELECT statement does *not* modify the contents of the database. Therefore, you do *not* usually need to perform transaction management for SELECT statements.

Executing static queries that modify data: SQL statements that modify data in a database table include the following:

- INSERT adds new rows to a database table.
- UPDATE modifies existing rows of a database table.
- DELETE removes rows from a database table.

To send one of these statements as a static query to the database for execution, specify a string representation of the statement as an argument to the `executeSQL()` method. For example, the following call to `executeSQL()` sends an INSERT of one row into the `abc` table of the database associated with the current connection:

```
connection.executeSQL("insert into abc values (1, 3, 6)");
```

Note: In the preceding code, the `connection` variable is a `CwDBConnection` object obtained from a previous call to the `getDBConnection()` method.

For an UPDATE or INSERT statement, you can determine the number of rows in the database table that have been modified or added with the `getUpdateCount()` method.

Important: Because the INSERT, UPDATE, and DELETE statements modify the contents of the database, it is good practice to assess the need for transaction management for these statements. For more information, see “Managing the transaction” on page 224.

Executing a static stored procedure: You can use the `executeSQL()` method to execute a stored-procedure call as long as *both* of the following conditions exist:

- The stored procedure does *not* use OUT parameters.
If the stored procedure uses an OUT parameter, you *must* use `executeStoredProcedure()` to execute it.
- The stored procedure is called only once.
The `executeSQL()` method does *not* save the prepared statement for the stored-procedure call. Therefore, if you call the same stored procedure more than once (for example, in a loop), use of `executeSQL()` can be slower than calling a method that does save the prepared statement: `executePreparedSQL()` or `executeStoredProcedure()`.

For more information, see “Executing stored procedures” on page 219.

Executing prepared queries

The “`executePreparedSQL()`” on page 401 method sends a prepared query to the database for execution. A **prepared query** is an SQL statement that is already precompiled into the executable form used by the database. The first time that `executePreparedSQL()` sends a query to the database, it sends the query as a string. The database receives this query, compiles it into an executable form by parsing the string, and executes the resulting SQL statement (just as it does for

`executeSQL()`). However, the database returns this compiled form of the SQL statement to `executePreparedSQL()`, which stores it in memory. This compiled SQL statement is called a **prepared statement**.

In subsequent executions of this same query, `executePreparedSQL()` first checks whether a prepared statement already exists for this query. If a prepared statement does exist, `executePreparedSQL()` sends it to the database instead of the query string. Subsequent executions of this query are more efficient because the database does not have to parse the string and create the prepared statement.

You can send the following kinds of SQL queries to a database with `executePreparedSQL()`:

- Queries that return data from the database (SELECT)
- Queries that modify data in the database (INSERT, UPDATE, DELETE)
- Queries that execute stored procedures defined in the database

Executing prepared queries that return data (SELECT): If you need to execute the same SELECT statement multiple times, use `executePreparedSQL()` to create a precompiled version of the statement. Keep the following in mind to prepare a SELECT statement:

- You can use parameters in this SELECT statement to pass specific information to each execution of the prepared statement. For an example of how to use parameters with a prepared statement, see Figure 87.
- When you execute a SELECT statement with `executePreparedSQL()`, you still use the same methods to access the returned data (Table 58 and Table 59). For more information, see “Executing static queries that return data (SELECT)” on page 214.

Executing prepared queries that modify data: If you need to execute the same INSERT, UPDATE, or DELETE statement multiple times, use `executePreparedSQL()` to create a precompiled version of the statement. The SQL statement that you reexecute does *not* need to be exactly the same in each time it executes to take advantage of the prepared statement. You can use parameters in the SQL statement to dynamically provide information to each statement execution.

The code fragment in Figure 87 inserts 50 rows into the `employee` table. The first time `executePreparedSQL()` is invoked, it sends the string version of the INSERT statement to the database, which parses it, executes it, and returns its executable form: a prepared statement. The next 49 times that this INSERT statement executes (assuming all INSERTs are successful), `executePreparedSQL()` recognizes that a prepared statement exists and sends this prepared statement to the database for execution.

```

CwDBConnection connection;
Vector argValues = new Vector();

argValues.setSize(2);

int emp_id = 1;
int emp_num = 2000;

for (int i = 1; i < 50; i++)
{
    argValues.set(0, new Integer(emp_id));
    argValues.set(1, new Integer(emp_num));

    try
    {
        // Send the INSERT statement to the database
        connection.executePreparedSQL(
            "insert into employee (employee_id, employee_number) values (?, ?)",
            argValues);

        // Increment the argument values
        emp_id++;
        emp_num++;
    }

    catch(CwDBSQLException e)
    {
        System.out.println(e.getMessage());
    }
}

```

Figure 87. Passing argument values to a prepared statement

Tip: Executing the prepared version of the INSERT statement usually improves application performance, although it does increase the application’s memory footprint.

When you reexecute an SQL statement that modifies the database, you must still handle transactions according to the transaction programming model. For more information, see “Managing the transaction” on page 224.

Note: To simplify the code in Figure 87 does *not* include transaction management.

Executing a prepared stored procedure: You can use the `executePreparedSQL()` method to execute a stored-procedure call as long as *both* of the following conditions exist:

- The stored procedure uses does *not* contain OUT parameters.
If the stored procedure uses an OUT parameter, you *must* use `executeStoredProcedure()` to execute it.
- The stored procedure is called more than once.
The `executePreparedSQL()` method saves the prepared statement for the stored-procedure call in memory. Therefore, if you call the stored procedure only once, use of `executePreparedSQL()` can use more memory than calling the stored procedure with `executeSQL()`, which does not save the prepared statement.

For more information, see “Executing stored procedures” on page 219.

Executing stored procedures

A **stored procedure** is a user-defined procedure that contains SQL statements and conditional logic. Stored procedures are stored in a database along with the data.

Note: When you create a new relationship, Relationship Designer creates a stored procedure to maintain each relationship table.

Table 60 shows the methods in the `CwDBConnection` class that call a stored procedure.

Table 60. *CwDBConnection* methods for calling a stored procedure

How to call the stored procedure	CwDBConnection method	Use
Send to the database a CALL statement to execute the stored procedure.	<code>executeSQL()</code>	To call a stored procedure that does <i>not</i> have OUT parameters and is executed <i>only once</i>
	" <code>executePreparedSQL()</code> " on page 401	To call a stored procedure that does <i>not</i> have OUT parameters and is executed <i>more than once</i>
Specify the name of the stored procedure and an array of its parameters to create a procedure call, which is sent to the database for execution.	<code>executeStoredProcedure()</code>	To call any stored procedure, including one with OUT parameters

Note: You can use JDBC methods to execute a stored procedure directly. However, the interface that the `CwDBConnection` class provides is simpler and it reuses database resources, which can increase the efficiency of execution. You can use of the methods in the `CwDBConnection` class to execute stored procedures.

A stored procedure can return data in the form of one or more rows. In this case, you use the same Java methods (such as `hasMoreRows()` and `nextRow()`) to access these returned rows in the query result as you do for data returned by a SELECT statement. For more information, see "Executing static queries that return data (SELECT)" on page 214.

As Table 60 shows, the choice of which method to use to call a stored procedure depends on:

- Whether the procedure provides any OUT parameters
An OUT parameter is a parameter through which the stored procedure returns a value to the calling code. If the stored procedure uses an OUT parameter, you *must* use `executeStoredProcedure()` to call the stored procedure.
- The number of times you call the stored procedure
The `executeStoredProcedure()` method saves the compiled version of the stored procedure. Therefore, if you call the same stored procedure more than once (for example, in a loop), use of `executeStoredProcedure()` can be faster than `executeSQL()` because the database can reuse the precompiled version.

The following sections describe how to use the `executeSQL()` and `executeStoredProcedure()` methods to call a stored procedure.

Calling stored procedures with no OUT parameters: To call a stored procedure that does *not* include any OUT parameters, you can use either of the following methods of `CwDBConnection`:

- The `executeSQL()` method sends a static stored-procedure call to the database. This procedure call is sent as a string to the database, which compiles it into a prepared statement before executing it. This prepared statement is *not* saved. Therefore, `executeSQL()` is useful for a stored procedure that only needs to be called once.

- The "executePreparedSQL()" on page 401 method sends a prepared stored-procedure call to the database.

In its first invocation, this procedure call is sent to the database, which creates the prepared statement and executes it. However, the database then sends this prepared statement back to executePreparedSQL(), which saves it in memory. Therefore, executePreparedSQL() is useful for a stored procedure that needs to be called more than once (for example, in a loop).

To call a stored procedure with one of these methods, specify as an argument to the method a string representation of the CALL statement that includes the stored procedure and any arguments. In Figure 88, the call to executeSQL() sends a CALL statement to execute the setOrderCurrDate() stored procedure.

```
connection.executeSQL("call setOrderCurrDate(345698)");
```

Figure 88. Calling a stored procedure with executeSQL()

In Figure 88, the connection variable is a CwDBConnection object obtained from a previous call to the getDBConnection() method. You can use executeSQL() to execute the setOrderCurrDate() stored procedure because its single argument is an IN parameter; that is, the value is *only* sent into the stored procedure. This stored procedure does *not* have any OUT parameters.

You can use the form of executeSQL() or executePreparedSQL() that accepts a parameter array to pass in argument values to the stored procedure. However, you *cannot* use these methods to call a stored procedure that uses an OUT parameter. To execute such a stored procedure, you *must* use executeStoredProcedure(). For more information, see "Calling stored procedures with executeStoredProcedure()" on page 221.

Note: Use an anonymous PL/SQL block if you plan on calling Oracle stored PL/SQL objects via ODBC using the CwDBConnection.executeSQL() method. The following is an acceptable format (the stored procedure name is myproc):

```
connection.executeSQL("begin myproc(...); end;");
```

Calling stored procedures with executeStoredProcedure(): The executeStoredProcedure() method can execute any stored procedure, including one that uses OUT parameters. This method saves the prepared statement for the stored-procedure call, just as the executePreparedSQL() method does. Therefore, executeStoredProcedure() can improve performance of a stored-procedure call that is executed multiple times.

To call a stored procedure with the executeStoredProcedure() method, you:

1. Specify as a String the name of the stored procedure to execute.
2. Build a Vector parameter array of CwDBStoredProcedureParam objects, which provide parameter information: the in/out parameter type and value of each stored-procedure parameter.

A **parameter** is a value you can send into or out of the stored procedure. The parameter's in/out type determines how the stored procedure uses the parameter value:

- An IN parameter is for *input only*: the stored procedure accepts the parameter value as input but does *not* use the parameter to return a value to the calling code.

- An OUT parameter is for *output only*: the stored procedure does *not* interpret the parameter value as input but uses the parameter to return a value to the calling code.
- An INOUT parameter is for both *input and output*: the stored procedure accepts the parameter value as input and uses the parameter to return a value to the calling code.

A `CwDBStoredProcedureParam` object describes a single parameter of a stored procedure. Table 61 shows the parameter information that a `CwDBStoredProcedureParam` object contains as well as the methods to retrieve and set this parameter information.

Table 61. Parameter information in a `CwDBStoredProcedureParam` object

Parameter information	<code>CwDBStoredProcedureParam</code> method
Parameter value	<code>getValue()</code>
Parameter in/out type	<code>getParamType()</code>

To pass parameters to a stored procedure with `executeStoredProcedure()`:

1. Create a `CwDBStoredProcedureParam` object to hold the parameter information. Use the `CwDBStoredProcedureParam()` constructor to create a new `CwDBStoredProcedureParam` object. To this constructor, pass the following parameter information to initialize the object:
 - Parameter in/out type specifies whether the parameter is an IN, INOUT, or OUT parameter.
 - Parameter value is a Java data type that contains the value to assign to the parameter. The `CwDBStoredProcedureParam` class provides many versions of its constructor to support the different data types that could be associated with the parameter value. For an OUT parameter, this parameter value can be a dummy value but the data type should correspond to the OUT parameter data type in the stored-procedure declaration.
2. Repeat step 1 for each stored-procedure parameter.
3. Create a `Vector` object with enough elements to hold all stored-procedure parameters.
4. Add the initialized `CwDBStoredProcedureParam` object to the parameter `Vector` object. Use the `addElement()` or `add()` method of the `Vector` class to add the `CwDBStoredProcedureParam` object.
5. Once you have created all `CwDBStoredProcedureParam` objects and added them to the `Vector` parameter array, pass this parameter array as the second argument to the `executeStoredProcedure()` method. The `executeStoredProcedure()` method sends the stored procedure and its parameters to the database for execution.

For example, suppose you have the `get_empno()` stored procedure defined in a database as follows:

```
create or replace procedure get_empno(emp_id IN number,
    emp_number OUT number) as
begin
    select emp_no into emp_number
    from emp
    where emp_id = 1;
end;
```

This `get_empno()` stored procedure has two parameters:

- The first parameter, `emp_id`, is an IN parameter.

Therefore, you must initialize its associated `CwDBStoredProcedureParam` object with an in/out type of `PARAM_IN`, as well as with the appropriate value to send into the stored procedure. Because `emp_id` is declared as the SQL `NUMBER` type (which holds an integer value), the parameter's value is of a Java Object that holds integer values: `Integer`.

- The second parameter, `emp_number`, is an OUT parameter.

For this parameter, create an *empty* `CwDBStoredProcedureParam` object to send into the stored procedure. You initialize this object with an in/out type of `PARAM_OUT`. However, you provide a dummy `Integer` value for this parameter. Once the stored procedure completes execution, you can obtain the returned value from this OUT parameter with the `getValue()` method.

Figure 89 executes the `get_empno()` stored procedure with the `executeStoredProcedure()` method to obtain the employee number for an employee id of 65:

```
CwDBConnection connectn = null;

try
{
    // Get database connection
    connectn = getDBConnection("CustomerDBPool");

    // Create parameter Vector
    Vector paramData = new Vector(2);

    // Create IN parameter for the employee id and add to parameter
    // vector
    paramData.add(
        new CwDBStoredProcedureParam(PARAM_IN, new Integer(65)));

    // Create dummy argument for OUT parameter and add to parameter
    // vector
    paramData.add(
        new CwDBStoredProcedureParam(PARAM_OUT, new Integer(0)));

    // Call the get_empno() stored procedure
    connectn.executeStoredProcedure("get_empno", paramData);

    // Get the result from the OUT parameter
    CwDBStoredProcedureParam outParam =
        (CwDBStoredProcedureParam) paramData.get(1);
    int emp_number = ((Integer) outParam.getValue()).intValue();
}
```

Figure 89. Executing the `get_empno()` stored procedure

Tip: The Java `Vector` object is a zero-based array. In the preceding code, to access the value for this OUT parameter from the `Vector` parameter array, the `get()` call specifies an index value of 1 because this `Vector` array is zero-based.

A stored procedure processes its parameters as SQL data types. Because SQL and Java data types are *not* identical, the `executeStoredProcedure()` method must convert a parameter value between these two data types. For an IN parameter, `executeStoredProcedure()` converts the parameter value from a Java data type to its SQL data type. For an OUT parameter, `executeStoredProcedure()` converts the parameter value from its SQL data type to a Java data type.

The `executeStoredProcedure()` method uses the JDBC data type internally to hold the parameter value sent to and from the stored procedure. JDBC defines a set of generic SQL type identifiers in the `java.sql.Types` class. These types represent the most commonly used SQL types. JDBC also provides standard mapping from JDBC types to Java data types. For example, a JDBC `INTEGER` is normally mapped to a Java `int` type. The `executeStoredProcedure()` method uses the mappings shown in Table 62.

Table 62. Mappings between Java and JDBC data types

Java data type	JDBC data type
<code>String</code>	<code>CHAR</code> , <code>VARCHAR</code> , or <code>LONGVARCHAR</code>
<code>Integer</code> , <code>int</code>	<code>INTEGER</code>
<code>Long</code>	<code>BIGINT</code>
<code>Float</code> , <code>float</code>	<code>REAL</code>
<code>Double</code> , <code>double</code>	<code>DOUBLE</code>
<code>java.math.BigDecimal</code>	<code>NUMERIC</code>
<code>Boolean</code> , <code>boolean</code>	<code>BIT</code>
<code>java.sql.Date</code>	<code>DATE</code>
<code>java.sql.Time</code>	<code>TIME</code>
<code>java.sql.Timestamp</code>	<code>TIMESTAMP</code>
<code>java.sql.Clob</code>	<code>CLOB</code>
<code>java.sql.Blob</code>	<code>BLOB</code>
<code>byte[]</code>	<code>BINARY</code> , <code>VARBINARY</code> , or <code>LONGVARBINARY</code>
<code>Array</code>	<code>ARRAY</code>
<code>Struct</code>	<code>STRUCT</code>

Managing the transaction

A **transaction** is a set of operational steps that execute as a unit. All SQL statements that execute within a transaction succeed or fail as a unit. This section provides the following information about managing transactions:

- “Determining the transaction programming model”
- “Specifying the transaction scope” on page 225

Determining the transaction programming model

The grouping of the database operation execution steps into transactions is called **transaction bracketing**. Associated with each connection is one of the following transaction programming models:

- Implicit transaction bracketing—database operations are part of an **implicit transaction**, which begins as soon as the connection is acquired and ends when the connection is released; transaction bracketing is implicitly managed by InterChange Server.
- Explicit transaction bracketing—database operations are part of an **explicit transaction**, whose beginning and end of each transaction is determined programmatically.

At runtime, a collaboration object determines which transaction programming model to use for each connection it acquires. By default, a collaboration object assumes that *all* connections it acquires use implicit transaction bracketing as their transaction programming model. You can override the default transaction programming model in any of the ways listed in Table 63.

Table 63. Overriding the transaction programming model for a connection

Transaction programming model to override	Action to take
To specify a different transaction programming model <i>for all connections</i> obtained by a particular collaboration object	Check or uncheck the Implicit Database Transaction box on the Collaboration Properties dialog of System Manager. For more information, see the <i>Implementation Guide for WebSphere InterChange Server</i> .
To specify a transaction programming model <i>for a particular connection</i>	<p>Provide a boolean value to indicate the desired transaction programming model (for this connection only) as the optional second argument to the <code>getDBConnection()</code> method.</p> <p>The following <code>getDBConnection()</code> call specifies explicit transaction bracketing for the connection obtained from the <code>ConnPool</code> connection pool:</p> <pre>conn = getDBConnection("ConnPool", false);</pre>

You can determine the current transaction programming model that connections will use with the `BaseCollaboration.implicitDBTransactionBracketing()` method, which returns a boolean value indicating whether the transaction programming model is implicit transaction bracketing.

Specifying the transaction scope

The connection's transaction programming model determines how the scope of the database transaction is specified. Therefore, this section provides the following information:

- "Transaction scope with implicit transaction bracketing"
- "Transaction scope with explicit transaction bracketing" on page 226

Transaction scope with implicit transaction bracketing: InterChange Server handles transaction management for all collaborations. All actions in the collaboration's business process are either completed as a unit or not completed. Therefore, InterChange Server handles the business process as a whole as a single implicit transaction. If any task fails, you choose how to handle the failed collaboration through the Unresolved Flow browser.

If the connection uses implicit transaction bracketing, InterChange Server also handles transaction management for operations performed on an external database, one associated with a connection from a connection pool. When a collaboration performs database operations, these database operations are part of the collaboration's business process. InterChange Server handles these database operations as an implicit transaction, which is subtransaction of the main transaction (the collaboration's business process). This database subtransaction begins as soon as the collaboration obtains the connection. ICS implicitly ends the subtransaction when execution of the collaboration completes.

The success or failure of this database subtransaction depends on the success or failure of the main transaction, as follows:

- If the collaboration is successful, InterChange Server commits the database subtransaction.
- If the collaboration fails, InterChange Server rolls back the database subtransaction. If this rollback fails, InterChange Server throws the `CwDBTransactionException` exception and logs an error.

When a collaboration invokes another collaboration directly, the first collaboration is called the parent, while the second one is called the child. When a parent collaboration calls a child collaboration, InterChange Server manages the transaction for the child collaboration separately. The success or failure of this child collaboration is independent of the success or failure of the parent collaboration. If the child collaboration fails, the parent collaboration can decide how to handle this failure. For example, it can decide it must fail as well, or it can decide to fix or ignore the situation and continue execution.

However, even though the success and failure of a child collaboration is independent of the parent collaboration, the same is not true of any implicit database transactions that the child might perform. If the child collaboration performs database operations through a database connection that uses implicit transaction bracketing, the child collaboration inherits the transaction of the parent collaboration. InterChange Server handles these database operations as a subtransaction of the parent collaboration. That is, the failure or success of the parent (or top-level) collaboration determines the final transactional state of the implicit database subtransaction in the child collaboration, as follows:

- If the parent collaboration is successful, InterChange Server commits the database subtransaction.
- If the parent collaboration fails, InterChange Server rolls back the database subtransaction.

InterChange Server does not commit or roll back the subtransaction until it knows the success or failure of the parent collaboration.

With this behavior, if the child collaboration failed and the parent collaboration chose to continue execution, InterChange Server would commit the implicit database transaction of the child collaboration.

Note: InterChange Server handles any explicit database subtransactions that the child collaboration performs and that are still active (i.e. those that have been performed through a database connection that uses explicit transaction bracketing but not been explicitly committed or rolled back in the child's collaboration template) in the same way as implicit database subtransactions.

This method of handling child transactions provides the collaboration developer with a means to perform a transactional join from child to parent without explicitly using join semantics. If instead, child database transactions were committed or rolled back at the child level, the developer could not correlate transactions that the child started (either explicitly or implicitly) with the global business process transaction that the parent started.

Note: Transactional collaborations use this same model for their parent and child database transactions.

Transaction scope with explicit transaction bracketing: If the connection uses explicit transaction bracketing, ICS expects the collaboration template to explicitly specify the scope of each database transaction. Explicit transaction bracketing is useful if you have some database work to perform that is independent of the success or failure of the collaboration. For example, if you need to perform auditing to indicate that certain tables were accessed, this audit needs to be performed regardless of whether the table accesses were successful or not. If you contain the auditing database operations in an explicit transaction, they are executed regardless of the success or failure of the collaboration.

Table 64 shows the methods in the `CwDBConnection` class that provide management of transaction boundaries for explicit transactions.

Table 64. CwDBConnection methods for explicit transaction management

Transaction-management task	CwDBConnection method
Begin a new transaction.	"beginTransaction()" on page 399
End the transaction, committing (saving) all changes made during the transaction to the database.	"commit()" on page 400
Determine if a transaction is currently active.	inTransaction()
End the transaction, rolling back (backing out) all changes made during the transaction.	rollback()

To specify transaction scope of an explicit transaction, follow these steps:

1. Mark the beginning of the transaction with a call to the `beginTransaction()` method.
2. Execute all SQL statements that must succeed or fail as a unit *between* this call to `beginTransaction()` and the end of the transaction.
3. End the transaction in either of two ways:
 - Call `commit()` to end the transaction successfully. All modifications that the SQL statements have made are *saved* in the database.
 - Call `rollback()` to end the transaction unsuccessfully. All modifications that the SQL statements have made are *backed out* of the database.

You can choose what conditions cause a transaction to fail. Test the condition and call `rollback()` if any failure condition is met. Otherwise, call `commit()` to end the transaction successfully.

Important: If you do *not* use `beginTransaction()` to specify the beginning of the explicit transaction, the database executes each SQL statement as a separate transaction. If you include `beginTransaction()` but do *not* specify the end of the database transaction with `commit()` or `rollback()` before the connection is released, InterChange Server implicitly ends the transaction based on the success of the collaboration. If the collaboration is successful, ICS commits this database transaction. If the collaboration is *not* successful, ICS implicitly rolls back the database transaction. Regardless of the success of the collaboration, ICS logs a warning.

The following code fragment updates three tables in the database associated with connections in the `CustDBConnPool`. If *all* these updates are successful, the code fragment commits these changes with the `commit()` method. If any transaction errors occur, a `CwDBTransactionException` exception results and the code fragment invokes the `rollback()` method.

```
CwDBConnection connection = getDBConnection("CustDBConnPool", false);

// Begin a transaction
connection.beginTransaction();

// Update several tables
try
{
    connection.executeSQL("update table1....");
    connection.executeSQL("update table2....");
    connection.executeSQL("update table3....");
}
```

```

        // Commit the transaction
        connection.commit();
    }

    catch (CwDBSQLException e)
    {
        // Roll back the transaction if an executeSQL() call throws
        // an exception
        connection.rollback();
    }

    // Release the database connection
    connection.release();

```

To determine whether a transaction is currently active, use the `inTransaction()` method.

Attention: Use the `beginTransaction()`, `commit()`, and `rollback()` methods *only* if the connection uses explicit transaction bracketing. If the connection uses implicit transaction bracketing, use of any of these methods results in a `CwDBTransactionException` exception.

Releasing a connection

Once a connection is released, it is returned to its connection pool, where it is available for use by other components. The way that a connection to the database is released depends on the transaction programming model. Therefore, this section provides the following information:

- “Releasing a connection with implicit transaction bracketing”
- “Releasing a connection with explicit transaction bracketing”

Releasing a connection with implicit transaction bracketing

ICS automatically releases a connection that uses implicit transaction bracketing once it has ended the database transaction. ICS does not end the database transaction until it determines the success or failure of the collaboration object; that is, ICS releases these connections when the collaboration finishes execution. If the collaboration executes successfully, ICS automatically commits any database transactions that are still active. If the collaboration execution fails (for instance, if an exception is thrown that is not handled with a catch statement), ICS automatically rolls back any transactions that are still active.

Releasing a connection with explicit transaction bracketing

For a connection that uses explicit transaction bracketing, the connection ends in either of the following cases:

- ICS automatically releases a connection that uses explicit transaction bracketing.
- You explicitly release a connection with the `release()` method of the `CwDBConnection` class.

You can use the `CwDBConnection.isActive()` method to determine whether a connection has been released. If the connection has been released, `isActive()` returns `false`, as the following code fragment shows:

```

if (connection.isActive())
    connection.release();

```

Attention: Do *not* use the `release()` method if a transaction is currently active. With implicit transaction bracketing, ICS does not end the database transaction until it determines the success or failure of the collaboration. Therefore, use of this method on a connection that uses

implicit transaction bracketing results in a `CwDBTransactionException` exception. If you do not handle this exception explicitly, it also results in an automatic rollback of the active transaction. You can use the `inTransaction()` method to determine whether a transaction is active. ICS automatically releases a connection regardless of the transaction programming model it uses. In most cases, you do not need to explicitly release the connection.

Chapter 10. Creating a message file

A collaboration uses certain methods to generate messages. There are two ways for a collaboration to generate message text visible to a user:

- The collaboration calls a method for message display and includes the message text as a parameter to the call.
- The collaboration calls the messaging method and the call contains a reference to an external message file that contains the message text.

Generally, it is better practice to design a collaboration to refer to a message file than to generate the text itself. Keeping messages in a centralized message file, rather than within individual collaborations, makes maintenance, administration, and internationalization easier.

It is good practice to create a message file for each collaboration template. Process Designer provides the Messages view to facilitate message creation. When InterChange Server starts a collaboration object, it attempts to load the associated message file into memory. It logs a warning if the message file is missing.

This chapter describes how message files work, as well as how to create and maintain them. It covers the following topics:

- “Operations that use the message file”
- “Creating a message file”
- “Message file: Name and location” on page 232
- “Explanations” on page 233
- “Message parameters” on page 233
- “Maintaining the file” on page 234

Operations that use the message file

A message file can hold the text for messages used in several types of operations. Table 65 lists the types of operations that use a message file and the methods of the BaseCollaboration class that perform those operations.

Table 65. Message-generating operations

Operation	Methods
Logging	BaseCollaboration.logInfo() BaseCollaboration.logError() BaseCollaboration.logWarning()
Tracing	BaseCollaboration.trace()
Raising exceptions	BaseCollaboration.raiseException()

Creating a message file

Perform the following steps to create a message file:

1. Ensure that Process Designer is open.
2. Click Template → Open Template Messages. The Template Messages window is displayed, as shown in Figure 90 on page 232.

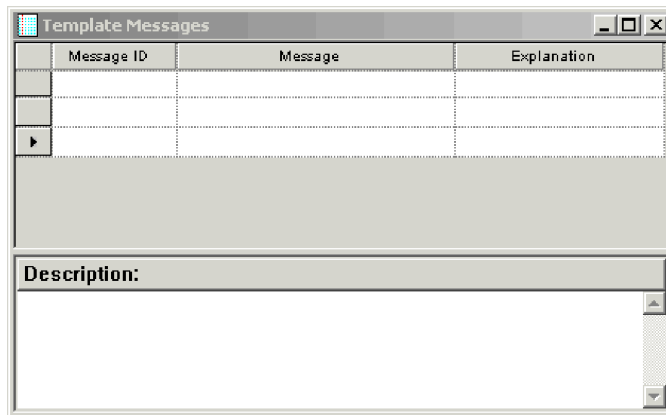


Figure 90. Template Messages window

3. For each message you create, do the following:
 - a. In the Message ID column, specify the message's unique identifier.
 - b. In the Message column, type the text of the message. This text is what the user sees when the message is displayed during runtime. You can include parameters in the message text to enable message reuse. See "Message parameters" on page 233.
 - c. Optionally, make the message self-documenting by adding an explanation in the Explanation column. See "Explanations" on page 233.
 - d. Optionally, add a description for the message in the Description pane. Text entered here is not visible to the user during runtime.

Message file: Name and location

The content of the collaboration message file is stored as part of the collaboration template. When you compile and deploy a collaboration, Process Designer extracts the message content and creates or updates the message file for runtime use. After compilation, the message file is located in the `\Templates\messages` directory of your Integration Component Library project in System Manager. After deployment, the message file is copied and placed in the `productDir\collaborations\messages` directory.

The name of the message file has the following format:

`CollaborationName.txt`

For example, when you compile and deploy a collaboration template named `SampleHello`, Process Designer creates a message file called `SampleHello.txt` and places it in the `collaborations\messages` subdirectory.

Important: Never make changes directly to a collaboration message file. Always use Process Designer to edit template messages.

The collaboration message file contains all text strings that the collaboration uses. These strings include those for logging, exception handling, and email operations.

Note: InterChange Server standards recommend that trace messages are *not* included in a collaboration message file because end users do not normally view them.

For an internationalized collaboration, it is important that these text strings are isolated into the collaboration message file so that this message file can be translated. The name of the translated collaboration message file must include the name of the associated locale (*CollaborationName_ll_TT.txt*).

In the preceding line, *ll* is the two-letter abbreviation for the locale (by convention in lowercase letters) and *TT* is the two-letter abbreviation for the territory (by convention in uppercase letters). For example, the version of the SampleHello collaboration's message file that contains U.S. English messages has the name SampleHello_en_US.txt

At runtime, the collaboration runtime environment locates the appropriate message file for the collaboration locale (inherited from InterChange Server) from the collaborations\messages subdirectory. For example, if the collaboration locale is U.S. English (en_US), the collaboration runtime environment retrieves messages from the *CollaborationName_en_US.txt* file.

For more information on how to internationalize the text strings of a collaboration, see "An internationalized collaboration" on page 63.

Explanations

Use the Explanation column in the Template Messages window to add a detailed explanation of each message to create self-documenting messages. Explanations are optional, but can improve the usability of your collaboration.

For example, suppose you have a message with the following text: Update failed. Destination application missing entry for {1} {2}. This message text does not provide enough detail for the user to easily remedy the error. In this example, adding a message explanation such as the following can significantly enhance the value of the message to the user:

An update request was sent to a connector, which successfully contacted the application. However, the application did not return data for the specified key attribute value.

For more examples of self-documenting messages, view the InterchangeSystem.txt file that is installed with WebSphere InterChange Server.

Reading message explanations

Explanations are not displayed with the message during runtime. Instead, the user must view them in the collaboration's message file.

If the InterChange Server log has been saved to a file, use Log Viewer to read any message explanations written to the log file. If the log is not saved to a file, message explanations must be viewed directly in the collaboration's message file.

Message parameters

It is not necessary to write separate messages for each possible situation. Instead, use parameters to represent values that change at runtime. The use of parameters allows each message to serve multiple situations and helps to keep the message file small.

A parameter always appears as a number surrounded by curly braces: *{number}*. For each parameter you want to add to the message, insert the number within curly braces into the text of the message, as follows:
message text *{number}* more message text.

The API method that is called to log the message must supply a value for each parameter. For example, consider the following message:

```
6  
Update failed. Destination application missing entry for {1} {2}
```

In the code fragment that sends this message, the following code appears:

```
logWarning(6, " CustomerID" , fromCust.getString("CustomerID"));
```

InterChange Server combines the parameter values in the `logWarning()` method call with the message in the log file and forms the message. Before writing the message to the log file, the server replaces the message parameters with the following values:

- Parameter 1 becomes the string " Customer ID".
- Parameter 2 becomes the value of the customer ID attribute in the `fromCust` business object.

The message then appears in the log file as follows:

```
Update failed. Destination application missing entry for CustomerID 101961
```

Because the message text takes the description of the missing entry and its ID as parameters, rather than including them as hardcoded strings, you can use the same message for any pair of missing attributes.

Maintaining the file

At a user site, an administrator can set up a procedure for filtering collaboration messages and using email or email pager to notify someone who can resolve problems. Because of this, it is important that the error numbers and the meanings associated with the numbers remain the same after the first release of a collaboration template. You can change the text associated with an error number, but you should not change the meaning of the text or reassign error numbers.

If you change the meanings associated with error numbers, make sure you document the change and notify users of the collaboration template.

You can change a collaboration's message file while the collaboration object is running. However, the changes do not take effect until the next time the collaboration object is started and the message file is read into memory. If InterChange Server goes down while collaborations are running, the server automatically reads into memory the message files for all collaborations that were previously running.

Part 3. Supported function blocks

Chapter 11. Business object function blocks

The function blocks in the General\APIs\Business Object folder and its subfolders provide basic functionality for working with business objects. Function blocks in the General\APIs\Business Object folder are based on methods in the Collaboration API's BusObj class. Function blocks in the General\APIs\Business Object\Array and General\APIs\Business Object\Constants folders correspond to Java arrays and constants of the class BusObj.

The following sections detail each function block.

Table 66. Summary of function blocks in the General\APIs\Business Object folder

Function Block	Page
Copy	238
Duplicate	239
Equals	240
Equal Keys	239
Exists	240
Get Boolean	241
Get Business Object	241
Get Business Object Array	242
Get Business Object Type	243
Get Double	243
Get Float	244
Get Int	245
Get Locale	245
Get Long	246
Get Long Text	246
Get Object	247
Get String	248
Get Verb	248
Is Blank	249
Is Business Object	249
Is Key	250
Is Null	250
Is Required	251
Iterate Children	251
Keys to String	251
New Business Object	252
Set Content	253
Set Default Attribute Values	253
Set Keys	254

Table 66. Summary of function blocks in the General\APIs\Business Object folder (continued)

Function Block	Page
Set Locale	254
Set Value	254
Set Value By Position	255
Set Value with Create	256
Set Verb	256
Set Verb with Create	257
Shallow Equals	257
To String	258
Valid Data	258

Table 67. Summary of function blocks in the General\APIs\Business Object\Array folder

Function block	Page
Get BusObj At	243
New Business Object Array	252
Set BusObj At	253
Size	258

Table 68. Summary of function blocks in the General\APIs\Business Object\Constants folder

Function block	Page
Verb:Create	259
Verb>Delete	259
Verb:Retrieve	260
Verb:Update	260

Copy

Copies all attribute values from the input business object.

Inputs

Copy to A BusObj object that represents the destination object for the copy operation.

Copy from A BusObj object that represents the business object to be copied.

Notes

The Copy function block copies the entire business object, including all child business objects and child business object arrays. This function block does not set a reference to the copied object. Instead, it clones all attributes (that is, it creates separate copies of the attributes).

Related information

This function block is based on the BusObj.copy() method. For more information, see “copy()” on page 368.

Duplicate

Creates a business object exactly like the original one.

Inputs

original The business object (a BusObj object) to be duplicated.

Output

Returns the duplicated business object.

Exceptions

The Duplicate function block can set the exception type of a CollaborationException object to ObjectException.

Notes

This function block makes a clone of the business object and returns it. You must explicitly assign the return value of this function block to a declared variable of type BusObj.

Related information

This function block is based on the BusObj.duplicate() method. For more information, see “duplicate()” on page 369.

Equal Keys

Compares the current business object’s key attribute values with those in the input business object to determine if they are equal.

Inputs

Business object 1

The first business object (a BusObj object) in the comparison.

Business object 2

The second business object (a BusObj object) in the comparison.

Output

Returns true if the values of all key attributes are the same; returns false if they are not the same.

Exceptions

The Equal Keys function block can set the following exception type for a CollaborationException exception:

- ObjectException—Set if the business object argument is invalid.

Notes

This function block performs a shallow comparison; it does not compare the keys in child business objects.

Related information

This function block is based on the `BusObj.equalKeys()` method. For more information, see “`equalKeys()`” on page 369.

Equals

Compares the attributes of two business objects (including child business objects) to determine if they are equal.

Inputs

Business object 1

The first business object (a `BusObj` object) in the comparison.

Business object 2

The second business object (a `BusObj` object) in the comparison.

Output

Returns true if the values of all attributes are the same; otherwise, returns false.

Exceptions

The Equals function block can set the following exception type for a `CollaborationException` exception:

- `ObjectException`—Set if the business object argument is invalid.

Notes

This function block compares this business object’s attribute values with those in the input business object. If the business objects are hierarchical, the comparison includes all attributes in the child business objects.

In the comparison, a null value is considered equivalent to any value to which it is compared and does not prevent a return of true.

Related information

This function block is based on the `BusObj.equals()` method. For more information, see “`equals()`” on page 370.

Exists

Checks for the existence of a business object attribute with a specified name.

Inputs

Business Object

The business object (a `BusObj` object).

Attribute

A String that specifies the name of the attribute whose existence you want to verify.

Output

Returns true if the attribute exists; otherwise, returns false.

Related information

This function block is based on the `BusObj.exists()` method. For more information, see “exists()” on page 372.

Get Boolean

Retrieves the value of a single attribute, as a boolean, from a business object.

Inputs

Business object

The business object (a `BusObj` object) in which the attribute exists.

Attribute

A String that specifies the name of the attribute.

Output

Returns the boolean value (true or false) of the specified attribute.

Exceptions

The Get Boolean function block can set the following exception type for a `CollaborationException` exception:

- `AttributeException`—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Boolean on an attribute that does not contain appropriate data.

Notes

The Get Boolean function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the `BusObj.getBoolean()` method. For more information, see “`getBoolean()`, `getDouble()`, `getFloat()`, `getInt()`, `getLong()`, `get()`, `getBusObj()`, `getBusObjArray()`, `getLongText()`, `getString()`” on page 372.

Get Business Object

Retrieves the value of a single attribute, as a `BusObj` object, from a business object.

Inputs

Business Object

The business object (a `BusObj` object).

Attribute

A String that specifies the name of the attribute you want to retrieve.

Output

Returns the value of the specified attribute as a `BusObj` object.

Exceptions

The Get Business Object function block can set the following exception type for a CollaborationException exception:

- **AttributeException**—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Business Object on an attribute that does not contain appropriate data.

Notes

The Get Business Object function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the BusObj.getBusObj() method. For more information, see “getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString()” on page 372.

Get Business Object Array

Retrieves the value of a single attribute, as a business object array, from a business object.

Inputs

Business object

The business object (a BusObj object).

Attribute

A String that specifies the name of the attribute you want to retrieve.

Output

Returns the value of the specified attribute as a business object array.

Exceptions

The Get Business Object Array function block can set the following exception type for a CollaborationException exception:

- **AttributeException**—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Business Object Array on an attribute that does not contain an array.

Notes

The Get Business Object Array function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the `BusObj.getBusObjArray()` method. For more information, see “`getBoolean()`, `getDouble()`, `getFloat()`, `getInt()`, `getLong()`, `get()`, `getBusObj()`, `getBusObjArray()`, `getLongText()`, `getString()`” on page 372.

Get Business Object Type

Retrieves the name of the business object definition on which the current business object was based.

Inputs

Business Object

The current business object (a `BusObj` object).

Output

Returns a `String` that contains the name of the business object definition.

Notes

The type of a business object is the name of the business object definition from which the business object was created.

Related information

This function block is based on the `BusObj.getType()` method. For more information, see “`getType()`” on page 375.

Get BusObj At

Retrieves the element at the specified index in a business object array.

Note: This function block is located in the `General\APIs\Business Object\Array` folder.

Inputs

Business object array

A `BusObj[]` object that represents the business object array.

Index

An integer that specifies the index location.

Output

Returns the business object located at the specified index.

Get Double

Retrieves the value of a single attribute, as a double, from a business object.

Inputs

Business object

The current business object (a `BusObj` object).

Attribute

A `String` that specifies the name of the attribute you want to retrieve.

Output

Returns the value of the specified attribute as a double data type.

Exceptions

The Get Double function block can set the following exception type for a CollaborationException exception:

- **AttributeException**—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Double on a String attribute that does not consist of digits.

Notes

The Get Double function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to attribute value in the source business object is *not* made to value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the BusObj.getDouble() method. For more information, see “getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString()” on page 372.

Get Float

Retrieves the value of a single attribute, as a float, from a business object.

Inputs

Business object

The current business object (a BusObj object).

Attribute

A String that specifies the name of the attribute you want to retrieve.

Output

Returns the value of the specified attribute as a float data type.

Exceptions

The Get Float function block can set the following exception type for a CollaborationException exception:

- **AttributeException**—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Float on a String attribute that does not consist of digits.

Notes

The Get Float function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the `BusObj.getFloat()` method. For more information, see “`getBoolean()`, `getDouble()`, `getFloat()`, `getInt()`, `getLong()`, `get()`, `getBusObj()`, `getBusObjArray()`, `getLongText()`, `getString()`” on page 372.

Get Int

Retrieves the value of a single attribute, as an integer, from a business object.

Inputs

Business object

The current business object (a `BusObj` object).

Attribute

A String that specifies the name of the attribute you want to retrieve.

Output

Returns the value of the specified attribute as an integer.

Exceptions

The Get Int function block can set the following exception type for a `CollaborationException` exception:

- `AttributeException`—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Int on a String attribute that does not consist of digits.

Notes

The Get Int function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the `BusObj.getInt()` method. For more information, see “`getBoolean()`, `getDouble()`, `getFloat()`, `getInt()`, `getLong()`, `get()`, `getBusObj()`, `getBusObjArray()`, `getLongText()`, `getString()`” on page 372.

Get Locale

Retrieves the locale associated with the business object’s data.

Inputs

Business object

The business object (a `BusObj` object).

Output

Returns a Java Locale object that contains information about the business object’s locale. This Locale object must be an instance of the `java.util.Locale` class.

Notes

The Get Locale function block returns the locale associated with the data in a business object. This locale is often different from the collaboration locale in which the collaboration is executing.

Related information

This function block is based on the `BusObj.getLocale()` method. For more information, see “`getLocale()`” on page 374.

Get Long

Retrieves the value of a single attribute, as a long data type, from a business object.

Inputs

Business object

The current business object (a `BusObj` object).

Attribute

A String that specifies the name of the attribute you want to retrieve.

Output

Returns the value of the specified attribute as a long data type.

Exceptions

The Get Long function block can set the following exception type for a `CollaborationException` exception:

- `AttributeException`—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Long on a String attribute that does not consist of digits.

Notes

The Get Long function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the `BusObj.getLong()` method. For more information, see “`getBoolean()`, `getDouble()`, `getFloat()`, `getInt()`, `getLong()`, `get()`, `getBusObj()`, `getBusObjArray()`, `getLongText()`, `getString()`” on page 372.

Get Long Text

Retrieves the value of a single attribute, as long text, from a business object.

Inputs

Business object

The current business object (a `BusObj` object).

Attribute A String that specifies the name of the attribute you want to retrieve.

Output

Returns the value of the specified attribute as a String.

Exceptions

The Get Long Text function block can set the following exception type for a CollaborationException exception:

- **AttributeException**—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Long Text on an attribute that contains data of an inappropriate type.

Notes

The Get Long Text function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

This function block returns a String object because the InterChange Server longtext type is a String object with no maximum size.

Related information

This function block is based on the BusObj.getLongText() method. For more information, see “getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString()” on page 372.

Get Object

Retrieves the value of a single attribute, as an object, from a business object.

Inputs

Business object

The current business object (a BusObj object).

Attribute

Either a String that specifies the name of the attribute, or an integer that specifies the ordinal position of the attribute in the business object’s attribute list.

Output

Returns the value of the specified attribute as an object.

Exceptions

The Get Object function block can set the following exception type for a CollaborationException exception:

- **AttributeException**—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get Object on an attribute that contains data of an inappropriate type.

Notes

The Get Object function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the BusObj.get() method. For more information, see “getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString()” on page 372.

Get String

Retrieves the value of a single attribute, as a String, from a business object.

Inputs

Business object

The current business object (a BusObj object).

Attribute

A String that specifies the name of the attribute you want to retrieve.

Output

Returns a String that contains the value of the specified attribute.

Exceptions

The Get String function block can set the following exception type for a CollaborationException exception:

- **AttributeException**—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration uses Get String on a nonexistent attribute.

Notes

The Get String function block retrieves an attribute value from the current business object and returns a copy of the attribute value. It does *not* return an object reference to this attribute in the source business object. Therefore, any change to the attribute value in the source business object is *not* made to the value that the function block returns. Each time this function block is used, it returns a new copy (clone) of the attribute.

Related information

This function block is based on the BusObj.getString() method. For more information, see “getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString()” on page 372.

Get Verb

Retrieves the verb for the current business object.

Inputs

Business object

The current business object (a BusObj object).

Output

Returns a String that contains the name of the business object's verb.

Notes

This function block is useful for scenarios that handle multiple types of incoming events. The first action node in a scenario calls Get Verb. The outgoing transition links from that action node then test the contents of the returned string, so that each outgoing transition link is the start of an execution path that handles one of the possible verbs.

Related information

This function block is based on the BusObj.getVerb() method. For more information, see "getVerb()" on page 375.

Is Blank

Determines whether the value of an attribute is set to a zero-length String.

Inputs

Business object

The current business object (a BusObj object).

Attribute

A String that specifies the name of the attribute you are querying.

Output

Returns true if the attribute value is a zero-length String; otherwise, returns false.

Notes

A zero-length string can be compared to the string "". It is different from a null, whose presence is detected by the Is Null function block.

If a collaboration needs to retrieve an attribute value and then do something with it, it can call Is Blank and Is Null to check that it has a value before retrieving the value.

Related information

This function block is based on the BusObj.isBlank() method. For more information, see "isBlank()" on page 376.

Is Business Object

Determines whether the value is a business object.

Inputs

Value

The object you want to query.

Output

Returns true if the value is a business object; otherwise, returns false.

Is Key

Determines whether a business object attribute is defined as a key attribute.

Inputs

Business object

The current business object (a BusObj object).

Attribute

A String that specifies the name of the attribute.

Output

Returns true if the attribute is a key attribute; otherwise, returns false.

Related information

This function block is based on the BusObj.isKey() method. For more information, see "isKey()" on page 376.

Is Null

Determines whether the value of a business object's attribute is null.

Inputs

Business object

The current business object (a BusObj object).

Attribute

A String that specifies the name of the attribute.

Output

Returns true if the attribute value is null; otherwise, returns false.

Notes

A null indicates no value, in contrast to a zero-length string value, which is detected by calling the Is Blank function block. Test an object with the Is Null function block before using it, because if the object is null, the operation can fail.

An attribute value can be null under these circumstances:

- The attribute value was explicitly set to null.
An attribute value can be set to null using the Set Value function block.
- The attribute value was never set.

When a collaboration uses the New Business Object function block to create a new business object, all attribute values are initialized to null. If the attribute value has not been set between the time of creation and the time the Is Null function block is called, the value is still null.

- The null was inserted during mapping.

When a collaboration processes a business object received from a connector, the mapping process might insert a null value. The mapping process converts the application-specific business object received from the connector to the generic

business object handled by the collaboration. For each attribute in the generic business object that has no equivalent in the application-specific object, the map inserts a null value.

Tip: Always call `IsNull` before performing an operation on an attribute that is a child business object or child business object array, because Java does not allow operations on null objects.

Related information

This function block is based on the `BusObj.isNull()` method. For more information, see “`isNull()`” on page 377.

Is Required

Determines whether a business object’s attribute is defined as a required attribute.

Inputs

Business object

The current business object (a `BusObj` object).

Attribute

A String that specifies the name of the attribute.

Output

Returns true if the attribute is required; otherwise, returns false.

Notes

If an attribute is defined as required, it must have a value and the value must not be a null.

Related information

This function block is based on the `BusObj.isRequired()` method. For more information, see “`isRequired()`” on page 378.

Iterate Children

Iterates through the child business object array.

Inputs

Business object

The current business object (a `BusObj` object).

Attribute

A String that specifies the name of the attribute.

Current index

An integer that specifies the current index.

Current element

The current element (a `BusObj` object).

Keys to String

Retrieves the values of a business object’s primary key attributes and returns them as a String.

Inputs

Business object

The current business object (a BusObj object).

Output

A String object containing all the key values in a business object, concatenated, and ordered by the ordinal value of the attributes.

Notes

The output from this function block contains the name of the attribute and its value. Multiple values are primary key attribute values, concatenated and separated by spaces. For example, if there is one primary key attribute, SS#, the output might be:

```
SS#=100408394
```

If the primary key attributes are FirstName and LastName, the output might be:

```
FirstName=Nina LastName=Silk
```

Related information

This function block is based on the BusObj.keysToString() method. For more information, see “keysToString()” on page 378.

New Business Object

Creates a new business object instance (BusObj) of the specified type.

Inputs

Type A String that specifies the type of business object you are creating.

Output

Returns a new business object of the type specified.

Related information

This function block is based on the Collaboration.BusObj() constructor.

New Business Object Array

Creates a new business object array.

Note: This function block is located in the General\APIs\Business Object\Array folder.

Inputs

Size An integer that specifies the size of the array.

Output

Returns a business object array of the specified size.

Set BusObj At

Sets the element at the specified index in a business object array.

Note: This function block is located in the General\APIs\Business Object\Array folder.

Inputs

Business object array

A BusObj[] object that represents the business object array.

Index

An integer that specifies the location of the element.

Business object

A BusObj object that represents the element to be set.

Set Content

Sets the contents of the current business object to another business object. The two business objects then own the content together. Changes made to one business object are reflected in the other business object.

Inputs

Business object

The current business object (a BusObj object).

Content

The business object (a BusObj object) whose content you want to use for the current business object.

Exceptions

The Set Content function block can set one of the following exception types for a CollaborationException exception:

- AttributeException—Set if an attribute access problem occurs.
- ObjectException—Set if the business object argument is invalid.

Related information

This function block is based on the BusObj.setContent() method. For more information, see the BusObj reference pages in the *Map Development Guide*.

Set Default Attribute Values

Sets all attributes of a business object to their default values.

Inputs

Business object

The current business object (a BusObj object).

Notes

A business object definition can include default values for attributes. The function block sets the values of this business object's attributes to the values specified as defaults in the definition.

Related information

This function block is based on the `BusObj.setDefaultAttrValues()` method. For more information, see “`setDefaultAttrValues()`” on page 380.

Set Keys

Sets the values of the current business object’s key attributes to the values of the key attributes in another business object.

Inputs

From business object

The business object (a `BusObj` object) whose key attribute values you want to use.

To business object

The current business object (a `BusObj` object) that is going to receive the key attribute values of the other business object.

Exceptions

The Set Keys function block can set one of the following exception types for a `CollaborationException` exception:

- `AttributeException`—Set if an attribute access problem occurs.
- `ObjectException`—Set if the business object argument is invalid.

Related information

This function block is based on the `BusObj.setKeys()` method. For more information, see “`setKeys()`” on page 380.

Set Locale

Sets the locale of the current business object.

Inputs

Business object

The business object (a `BusObj` object) whose locale you want to set.

Locale

The Java Locale object that contains the information about the locale to assign to the business object. This Locale object must be an instance of the `java.util.Locale` class.

Notes

The Set Locale function block assigns a locale to the data associated with a business object. This locale might be different from the collaboration locale in which the collaboration executes.

Related information

This function block is based on the `BusObj.setLocale()` method. For more information, see “`setLocale()`” on page 381.

Set Value

Sets a business object’s attribute to a specified value of a particular data type.

Inputs

Business object

The current business object (a BusObj object).

Attribute

A String that specifies the name of the attribute you want to set.

Value

The value for the attribute. Must be of the appropriate type (boolean, double, float, int, long, Object, String, or BusObj) for the attribute.

Exceptions

The Set Value function block can set the following exception type for a CollaborationException exception:

- AttributeException—Set if an attribute access problem occurs.

Notes

The Set Value function block sets an attribute value in the current business object. It sets an object reference when it assigns the value to the attribute. It does *not* clone the attribute value from the source business object. Therefore, any changes to value in the source business object are also made to the attribute in the business object that calls the Set Value function block.

The Set Value function block provides the following forms:

- The first form sets a value of the type specified by the function block's Value input. Use this form to set attributes with specific basic or InterChange Server-defined data types.
- The second form sets the value of an attribute of *any* type. You can send in any data type as the attribute value because the Value input is of type Object. For example, to set an attribute that is of BusObj or LongText object type, use the BusObj or LongText object for the Value input.

Related information

This function block is based on the BusObj.set() method. For more information, see "set()" on page 379.

Set Value By Position

Sets a business object's attribute to a specified value based on the attribute's position.

Inputs

Business object

The current business object (a BusObj object).

Position

An integer that specifies the ordinal position of an attribute in the business object's attribute list.

Value

The value for the attribute. Must be of the appropriate type (Object, String, or BusObj).

Exceptions

The Set Value By Position function block can set the following exception type for a CollaborationException exception:

- `AttributeException`—Set if an attribute access problem occurs.

Notes

The acceptable values for this function block are limited to the `Object`, `String`, and `BusObj` data types.

Related information

This function block is based on the `BusObj.set()` method. For more information, see “`set()`” on page 379.

Set Value with Create

Sets the business object’s attribute to a specified value of a particular data type, creating an object for the value if one does not already exist.

Inputs

Business object

The current business object (a `BusObj` object).

Attribute

A `String` that specifies the name of the attribute you want to set.

Verb

A `String` the specifies the verb `Create`.

Exceptions

The `Set Value with Create` function block can set the following exception type for a `CollaborationException` exception:

- `AttributeException`—Set if an attribute access problem occurs.

Notes

If the object provided is a `BusObj` and the target attribute contains multi-cardinality child business object, the `BusObj` is appended to the `BusObjArray` as its last element. If the target attribute contains a `BusObj`, however, this business object replaces the previous value.

Related information

This function block is based on the `BusObj.setWithCreate()` method. For more information, see “`setWithCreate()`” on page 382.

Set Verb

Sets the verb of a business object.

Inputs

Business object

The current business object (a `BusObj` object).

Verb

A `String` that specifies the verb to be used with the business object.

Notes

This function block is used in mapping business objects. Do not use the `Set Verb` function block in a collaboration template, where you must set the verb of an outgoing business object interactively by filling in the properties of a service call.

Related information

This function block is based on the `BusObj.setVerb()` method. For more information, see “`setVerb()`” on page 381.

Set Verb with Create

Sets the verb of a child business object, creating the child business object if one does not already exist.

Inputs

Business object

The current business object (a `BusObj` object).

Attribute

A String that specifies the name of the attribute.

Verb

A String that specifies the verb `Create`.

Exceptions

The Set Verb with Create function block can set the following exception type for a `CollaborationException` exception:

- `AttributeException`—Set if an attribute access problem occurs.

Notes

If the attribute specified by the Attribute input is of type `BusObj` and it is null, the new instance of that child business object is created and its verb set to the value of the Verb input. If the instance of this child business object already exists, only its verb is set. If the child business object is of multiple cardinality, the Attribute input must specify the subscript.

Related information

This function block is based on the `BusObj.setVerbWithCreate()` method.

Shallow Equals

Compares the values of two business objects, excluding child business objects, to determine whether they are equal.

Inputs

Business object 1

The first business object (a `BusObj` object) you are comparing.

Business object 2

The second business object (a `BusObj` object) you are comparing.

Output

Returns true if the values of all attributes are the same; otherwise, returns false.

Exceptions

The Shallow Equals function block can set the following exception type for a `CollaborationException` exception:

- `ObjectException`—Set if the business object argument is invalid.

Related information

This function block is based on the `BusObj.equalsShallow()` method. For more information, see “`equalsShallow()`” on page 371.

Size

Retrieves the size of the business object array.

Note: This function block is located in the `General\APIs\Business Object\Array` folder.

Inputs

Business object array

A `BusObj[]` object that represents the business object array.

Output

Returns an integer that specifies the size of the array.

To String

Returns the values of all attributes in a business object as a `String`.

Inputs

Business object

The current business object (a `BusObj` object).

Output

A `String` object containing all attribute values contained in a business object.

Notes

The string that results from a call to this method is similar to the following example:

```
Name: GenEmployee  
Verb: Create  
Type: AfterImage  
Attributes: (Name, Type, Value)
```

```
LastName:String, Davis  
FirstName:String, Miles  
SS#:String, 041-33-8989  
Salary:Float, 15.00  
ObjectEventId:String, MyConnector_922323619411_1
```

Related information

This function block is based on the `BusObj.toString()` method. For more information, see “`toString()`” on page 382.

Valid Data

Determines whether the specified value is a valid data type for a specified attribute.

Inputs

Business object

The current business object (a BusObj object).

Attribute

A String that specifies the name of the attribute.

Value

The value for the attribute. Can be of type Object, BusObj, BusObjArray, String, long, int, double, float, or boolean.

Output

Returns true if the specified value is a valid data type; otherwise, returns false.

Notes

Checks the compatibility of the value passed in with the target attribute (as specified by the Attribute input). These are the criteria:

for primitive types (String, long, int, double, float, boolean)	the value must be convertible to the data type of the attribute
for a BusObj	the value must have the same type as that of the target attribute
for a BusObjArray	the value must point to a BusObj or BusObjArray with the same (business object definition) type as that of the attribute
for an Object	the value must be of type String, BusObj, or BusObjArray. The corresponding validation rules are then applied.

Related information

This function block is based on the BusObj.validData() method. For more information, see “validData()” on page 383.

Verb:Create

The business object verb Create.

Note: This function block is located in the General\APIs\Business Object\Constants folder.

Output

Returns a String that contains the verb Create.

Verb>Delete

The business object verb Delete.

Note: This function block is located in the General\APIs\Business Object\Constants folder.

Output

Returns a String that contains the verb Delete.

Verb:Retrieve

The business object verb Retrieve.

Note: This function block is located in the General\APIs\Business Object\Constants folder.

Output

Returns a String that contains the verb Retrieve.

Verb:Update

The business object verb Update.

Note: This function block is located in the General\APIs\Business Object\Constants folder.

Output

Returns a String that contains the verb Update.

Chapter 12. Business object array function blocks

The function blocks in the General\APIs\Business Object Array folder provide basic functionality for working with business object arrays. These function blocks are based on the Collaboration API's BusObjArray class.

The following sections detail each function block.

Table 69. Summary of function blocks in the General\APIs\Business Object Array folder

Function block	Page
Add Element	261
Duplicate	262
Equals	262
Get Element At	263
Get Elements	263
Get Last Index	263
Is Business Object Array	264
Max Attribute Value	264
Max Business Object Array	265
Max Business Objects	266
Min Attribute Value	267
Min Business Object Array	267
Min Business Objects	268
Remove All Elements	269
Remove Element	269
Remove Element At	270
Set Element At	270
Size	271
Sum	271
Swap	272
To String	272

Add Element

Adds a business object to the current business object array.

Inputs

Business object array

The business object array (specified as a BusObjArray object).

Element

The business object (specified as a BusObj object) you want to add.

Exceptions

The Add Element function block can set the following exception type for a CollaborationException exception:

- **AttributeException**—Set if the element is not valid.

Related information

This function block is based on the BusObjArray.addElement() method. For more information, see “addElement()” on page 386.

Duplicate

Creates a business object array exactly like the original one.

Inputs

Original business object array

The business object array (specified as a BusObjArray object) you want to duplicate.

Output

Returns the duplicate business object array.

Related information

This function block is based on the BusObjArray.duplicate() method. For more information, see “duplicate()” on page 386.

Equals

Compares two business object arrays to determine whether they are equal.

Inputs

Business object array 1

The first business object array (specified as a BusObjArray object) you want to compare.

Business object array 2

The second business object array (specified as a BusObjArray object) you want to compare.

Output

Returns true if the arrays are equal; otherwise, returns false.

Notes

The comparison between the two business object arrays checks the number of elements and their attribute values.

Related information

This function block is based on the BusObjArray.equals() method. For more information, see “equals()” on page 387.

Get Element At

Retrieves a single business object from an array by specifying that object's position in the array.

Inputs

Business object array

The business object array (specified as a `BusObjArray` object).

Index

An integer that specifies the index location.

Output

Returns the specified business object.

Exceptions

The Get Element At function block can set the following exception type for a `CollaborationException` exception:

- `AttributeException`—Set if the element is not valid.

Related information

This function block is based on the `BusObjArray.elementAt()` method. For more information, see “`elementAt()`” on page 387.

Get Elements

Retrieves the contents of the business object array.

Inputs

Business object array

The business object array (specified as a `BusObjArray` object).

Output

Returns the business objects in the array.

Exceptions

The Get Elements function block can set the following exception type for a `CollaborationException` exception:

- `ObjectException`—Set if one of the elements is not valid.

Related information

This function block is based on the `BusObjArray.getElements()` method. For more information, see “`getElements()`” on page 387.

Get Last Index

Retrieves the last available index from a business object array.

Inputs

Business object array

The business object array (specified as a `BusObjArray` object).

Output

Returns the last index, as an integer.

Notes

Although the Get Last Index function block performs a similar function to the Size function block, there is one important difference to note: Size returns a value that is 1 greater than the value returned by Get Last Index. This discrepancy occurs because the BusObjArray object used as input is a zero relative array.

Related information

This function block is based on the BusObjArray.getLastIndex() method. For more information, see “getLastIndex()” on page 388.

Is Business Object Array

Determines whether an object is a business object array (BusObjArray).

Inputs

Value The name of the object to be tested.

Output

Returns true if the value is a business object array; otherwise, returns false.

Max Attribute Value

Retrieves the maximum values for the specified attribute among all elements in the business object array.

Inputs

Business object array
The business object array (specified as a BusObjArray object).

Attribute A String that specifies the attribute name.

Output

Returns a String that contains the maximum value for the specified attribute.

Exceptions

The Max Attribute Value function block can throw the following exceptions, both of which are subclassed from CollaborationException :

- UnknownAttributeException—When the specified attribute is not a valid attribute in the business objects passed in.
- UnsupportedAttributeTypeException—When the type of the specified attribute is not one of the supported attribute types listed in the Notes section.

For each of these exceptions, the Max Attribute Value function block can set the AttributeException exception type.

Notes

The Max Attribute Value function block looks for the maximum value for the specified attribute among the business objects in this BusObjArray. For example, if

three employee objects are used, and the attribute is Salary which is of type Float, it will return the string representing the largest salary.

If the value of the specified attribute for an element in BusObjArray is null, then that element is ignored. If the value of the specified attribute is null for all elements, then null is returned.

When the attribute type is of type String, Max Attribute Value returns the attribute value that is the longest string lexically.

Related information

This function block is based on the BusObjArray.max() method. For more information, see “max()” on page 388.

Max Business Object Array

Returns the business objects that have the maximum value for the specified attribute, as a business object array (BusObjArray object).

Inputs

Business object array

The business object array (specified as a BusObjArray object).

Attribute

A String that specifies the attribute name.

Output

A list of business objects in the form of a BusObjArray object.

Exceptions

The Max Business Object Array function block can throw the following exceptions, both of which are subclassed from CollaborationException :

- **UnknownAttributeException**—When the specified attribute is not a valid attribute in the business objects passed in.
- **UnsupportedAttributeTypeException**—When the type of the specified attribute is not one of the supported attribute types listed in the Notes section.

For each of these exceptions, the Max Business Object Array function block can set the AttributeException exception type.

Notes

The Max Business Object Array function block finds one or more business objects with the maximum value for the specified attribute, and returns these business objects in a BusObjArray object.

For example, suppose that this is a business object array containing Employee business objects and that the input argument is the attribute Salary, a Float. The function block determines the largest value for Salary in all the Employee business objects and returns the business object that contains that value. If multiple business objects have that largest Salary value, the function block returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type String, the function block returns the longest string lexically.

Related information

This function block is based on the `BusObjArray.maxBusObjArray()` method. For more information, see “`maxBusObjArray()`” on page 389.

Max Business Objects

Returns the business objects that have the maximum value for the specified attribute, as an array of `BusObj` objects.

Inputs

Business object array

The business object array (specified as a `BusObjArray` object).

Attribute

A String that specifies the attribute name.

Output

A list of business objects in the form of a `BusObj[]` object.

Exceptions

The Max Business Objects function block can throw the following exceptions, both of which are subclassed from `CollaborationException` :

- `UnknownAttributeException`—When the specified attribute is not a valid attribute in the business objects passed in.
- `UnsupportedAttributeTypeException`—When the type of the specified attribute is not one of the supported attribute types listed in the Notes section.

For each of these exceptions, the Max Business Objects function block can set the `AttributeException` exception type.

Notes

The Max Business Objects function block finds one or more business objects with the maximum value for the specified attribute, and returns these business objects as an array of `BusObj` objects.

For example, suppose that this is a business object array containing `Employee` business objects and that the input argument is the attribute `Salary`, a `Float`. The function block determines the largest value for `Salary` in all the `Employee` business objects and returns the business object that contains that value. If multiple business objects have that largest `Salary` value, the function block returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type String, the function block returns the longest string lexically.

Related information

This function block is based on the `BusObjArray.maxBusObjs()` method. For more information, see “`maxBusObjs()`” on page 390.

Min Attribute Value

Retrieves the minimum value for the specified attribute among all the elements in the business object array.

Inputs

Business object array

The business object array (specified as a `BusObjArray` object).

Attribute

A String that specifies the attribute name.

Output

Returns a String that contains the maximum value for the specified attribute.

Exceptions

The Min Attribute Value function block can throw the following exceptions, both of which are subclassed from `CollaborationException` :

- `UnknownAttributeException`—When the specified attribute is not a valid attribute in the business objects passed in.
- `UnsupportedAttributeTypeException`—When the type of the specified attribute is not one of the supported attribute types listed in the Notes section.

For each of these exceptions, the Min Attribute Value function block can set the `AttributeException` exception type.

Notes

The Min Attribute Value function block looks for the minimum value for the specified attribute among the business objects in this business object array.

For example, suppose that this is a business object array containing `Employee` business objects and that the input argument is the attribute `Salary`, a `Float`. The function block determines the smallest value for `Salary` in all the `Employee` business objects and returns the business object that contains that value. If multiple business objects have that lowest `Salary` value, the function block returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type `String`, the function block returns the shortest string lexically.

Related information

This function block is based on the `BusObjArray.min()` method. For more information, see “`min()`” on page 391.

Min Business Object Array

Returns the business objects that have the minimum value for the specified attribute, as a business object array (`BusObjArray` object).

Inputs

Business object array

The business object array (specified as a `BusObjArray` object).

Attribute A String that specifies the attribute name.

Output

A list of business objects in the form of a `BusObjArray` object.

Exceptions

The Min Business Object Array function block can throw the following exceptions, both of which are subclassed from `CollaborationException` :

- `UnknownAttributeException`—When the specified attribute is not a valid attribute in the business objects passed in.
- `UnsupportedAttributeTypeException`—When the type of the specified attribute is not one of the supported attribute types listed in the Notes section.

For each of these exceptions, the Min Business Object Array function block can set the `AttributeException` exception type.

Notes

The Min Business Object Array function block finds one or more business objects with the minimum value for the specified attribute, and returns these business objects in a `BusObjArray` object.

For example, suppose that this is a business object array containing `Employee` business objects and that the input argument is the attribute `Salary`, a `Float`. The function block determines the smallest value for `Salary` in all the `Employee` business objects and returns the business object that contains that value. If multiple business objects have that smallest `Salary` value, the function block returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type `String`, the function block returns the shortest string lexically.

Related information

This function block is based on the `BusObjArray.minBusObjArray()` method. For more information, see “`minBusObjArray()`” on page 392.

Min Business Objects

Returns the business objects that have the minimum value for the specified attribute, as an array of `BusObj` objects.

Inputs

Business object array

The business object array (specified as a `BusObjArray` object).

Attribute A String that specifies the attribute name.

Output

A list of business objects in the form of a `BusObj[]` object.

Exceptions

The Min Business Objects function block can throw the following exceptions, both of which are subclassed from `CollaborationException` :

- `UnknownAttributeException`—When the specified attribute is not a valid attribute in the business objects passed in.
- `UnsupportedAttributeTypeException`—When the type of the specified attribute is not one of the supported attribute types listed in the Notes section.

For each of these exceptions, the Min Business Objects function block can set the `AttributeException` exception type.

Notes

The Min Business Objects function block finds one or more business objects with the maximum value for the specified attribute, and returns these business objects as an array of `BusObj` objects.

For example, suppose that this is a business object array containing `Employee` business objects and that the input argument is the attribute `Salary`, a `Float`. The function block determines the smallest value for `Salary` in all the `Employee` business objects and returns the business object that contains that value. If multiple business objects have that smallest `Salary` value, the function block returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type `String`, the function block returns the shortest string lexically.

Related information

This function block is based on the `BusObjArray.minBusObjs()` method. For more information, see “`minBusObjs()`” on page 393.

Remove All Elements

Removes all elements from the business object array.

Inputs

Business object array

The business object array (`BusObjArray`) from which the elements are going to be removed.

Related information

This function block is based on the `BusObjArray.removeAllElements()` method. For more information, see “`removeAllElements()`” on page 394.

Remove Element

Removes a business object element from a business object array.

Inputs

Business object array

The business object array (BusObjArray) from which the element is going to be removed.

Element

The business object (BusObj) element to remove from the array.

Exceptions

The Remove Element function block can set the following exception type for a CollaborationException exception:

- AttributeException—Set if the element is not valid.

Notes

After you delete an element from the array, the array resizes, changing the indexes of existing elements.

Related information

This function block is based on the BusObjArray.removeElement() method. For more information, see “removeElement()” on page 394.

Remove Element At

Removes a business object element from a particular position in the business object array.

Inputs

Business object array

The business object array (BusObjArray) from which the element is going to be removed.

Index

The index position (specified as an integer) of the element to remove.

Exceptions

The Remove Element At function block can set the following exception type for a CollaborationException exception:

- AttributeException—Set if the element is not valid.

Notes

After an element is removed from the array, the array resizes, possibly changing the indexes of existing elements.

Related information

This function block is based on the BusObjArray.removeElementAt() method. For more information, see “removeElementAt()” on page 394.

Set Element At

Sets the value of a business object in a business object array.

Inputs

Business object array

The business object array (specified as an object of type `BusObjArray`) in which the element's value is going to be set.

Element

The business object element (specified as an object of type `BusObj`) whose value you are going to set.

Index

The index position (specified as an integer) of the business object element to set.

Exceptions

The Set Element At function block can set the following exception type for a `CollaborationException` exception:

- `AttributeException`—Set if the element is not valid.

Notes

This function block sets the values of the business object at a specified array position to the values of an input business object.

Related information

This function block is based on the `BusObjArray.setElementAt()` method. For more information, see “`setElementAt()`” on page 395.

Size

Determines the number of elements in a business object array.

Inputs

Business object array

The business object array (specified as an object of type `BusObjArray`) whose size you want to determine.

Output

Returns an integer that specifies the number of elements in the array.

Related information

This function block is based on the `BusObjArray.size()` method. For more information, see “`size()`” on page 395.

Sum

Adds the values of the specified attribute for all business objects in this business object array.

Inputs

Business object array

The business object array (specified as an object of type `BusObjArray`).

Attribute

A String that specifies the name of the attribute.

Output

Returns the sum (as a double data type) of the specified attribute from the list of the business objects.

Exceptions

The Sum function block can throw the following exceptions, both of which are subclassed from `CollaborationException` :

- `UnknownAttributeException`—When the specified attribute is not a valid attribute in the business objects passed in.
- `UnsupportedAttributeTypeException`—When the type of the specified attribute is not one of the supported attribute types listed in the note section.

For each of these exceptions, the Sum function block can set the `AttributeException` exception type.

Related information

This function block is based on the `BusObjArray.sum()` method. For more information, see “`sum()`” on page 396.

Swap

Reverses the positions of two business objects in the business object array.

Inputs

Business object array

The business object array (specified as an object of type `BusObjArray`).

Index 1 An integer that specifies the position of the first business object.

Index 2 An integer that specifies the position of the second business object.

Related information

This function block is based on the `BusObjArray.swap()` method. For more information, see “`swap()`” on page 396.

To String

Retrieves the values in the business object array and returns them as a single `String`.

Inputs

Business object array

The business object array (specified as an object of type `BusObjArray`).

Output

Returns a `String` that contains all of the values in the business object array.

Related information

This function block is based on the `BusObjArray.toString()` method. For more information, see “`toString()`” on page 397.

Chapter 13. Collaboration template function blocks

The collaboration template function blocks provide basic functionality for operating on collaboration objects. These function blocks are organized into the following folders:

- General\APIs\Collaboration Template—Used to work with collaboration objects.
- General\APIs\Collaboration Template\Exception—Used to create new exception objects within a collaboration template.
- General\APIs\Collaboration Template\Exception\Constants—Used to represent specific exception types within a collaboration exception object.

The following sections provide more information about each of the collaboration template function blocks.

Table 70. Summary of collaboration template function blocks

Location	Function block	Page
General\APIs\Collaboration Template	Get Locale	274
	Get Message	274
	Get Message with Parameter	275
	Get Name	275
	Get Property	276
	Get Property Array	276
	Implicit DB Bracketing	276
	Is Trace Enabled	277
	Property Exists	278
	Send Email	283
General\APIs\Collaboration Template\Exception	Raise Collaboration Exception	278
	Raise Collaboration Exception 1	279
	Raise Collaboration Exception 2	280
	Raise Collaboration Exception 3	280
	Raise Collaboration Exception 4	281
	Raise Collaboration Exception 5	282
	Raise Collaboration Exception with Parameter	282
General\APIs\Collaboration Template\Exception\Constants	AnyException	274
	AttributeException	274
	JavaException	277
	ObjectException	277
	OperationException	278
	ServiceCallException	284
	SystemException	284
	TransactionException	284

AnyException

A constant that represents any type of exception.

Note: This function block is located in the General\APIs\Collaboration Template\Exception\Constants folder.

Output

Returns a String with the value "AnyException".

AttributeException

A constant that represents an attribute access problem (for example, if a collaboration uses the Get Double function block for a String-based attribute, or used the Get String function block on a nonexistent attribute).

Note: This function block is located in the General\APIs\Collaboration Template\Exception\Constants folder.

Output

Returns a String with the value "AttributeException".

Get Locale

Retrieves the collaboration locale for the current collaboration object.

Inputs

Collaboration The current collaboration object.

Output

Returns a Java Locale object that contains the language and country codes of the collaboration locale. This Locale object must be an instance of the java.util.Locale class.

Notes

The Get Locale function block returns the locale of the current flow. This flow locale is the locale associated with the collaboration object's triggering business object.

Related information

This function block is based on the BaseCollaboration.getLocale() method. For more information, see "getLocale()" on page 349.

Get Message

Retrieves a message from the collaboration message file.

Inputs

Collaboration The current collaboration object.

ID An integer that specifies the message number of a message in the collaboration's message file. The message file is indexed by message number.

Output

Returns a String object that contains the text for the message specified by the ID input.

Related information

This function block is based on the BaseCollaboration.getMessage() method. For more information, see "getMessage()" on page 350.

Get Message with Parameter

Retrieves a message from the collaboration message file.

Inputs

Collaboration The current collaboration object.

ID An integer that specifies the message number of a message in the collaboration's message file. The message file is indexed by message number.

Parameters An array of message-parameter values. Each is sequentially resolved to a parameter in the message text. Within the message (in the collaboration message file), message parameters are indicated by integers enclosed by braces; for example, {1}.

Output

Returns a String object that contains the text for the message identified by the ID and Parameters inputs.

Notes

The Get Message with Parameter function block takes a message number and an array of message-parameter values. It retrieves the associated message from the collaboration message file, replaces its message parameters with the objects in the parameter array, and returns the resulting message as a String object.

Related information

This function block is based on the BaseCollaboration.getMessage() method. For more information, see "getMessage()" on page 350.

Get Name

Retrieves the name of this collaboration object.

Inputs

Collaboration The current collaboration object.

Output

Returns a String that contains the name of the current collaboration object.

Related information

This function block is based on the `BaseCollaboration.getName()` method. For more information, see “`getName()`” on page 351.

Get Property

Retrieves the value of a collaboration configuration property.

Inputs

Collaboration The current collaboration object.

Property name

A String that specifies the collaboration configuration property you want to query.

Output

Returns a String that contains the value of the specified collaboration configuration property.

Get Property Array

Retrieves the value of a multi-element collaboration configuration property.

Inputs

Collaboration The current collaboration object.

Property name

A String that specifies the collaboration configuration property you want to query.

Output

Returns an array of String objects; each String object in the array contains the value for one element of the collaboration configuration property.

Implicit DB Bracketing

Retrieves the transaction programming model that the collaboration object uses for any connection it obtains.

Inputs

Collaboration The current collaboration object.

Output

Returns a boolean value to indicate the transaction programming model to be used in all database connections.

- A value of `true` indicates that all connections use *implicit* transaction bracketing.
- A value of `false` indicates that all connections use *explicit* transaction bracketing.

Notes

The Implicit DB Bracketing function block returns a boolean value to indicate which transaction programming model the collaboration object assumes is used by

all connections that it obtains. This function block is useful before obtaining a connection to see whether the current transaction programming model is appropriate for that connection.

Note: You can override the transaction programming model for a particular connection with the Get Database Connection function block (available in the General\APIs\Database Connection folder).

Related information

This function block is based on the `BaseCollaboration.implicitDBTransactionBracketing()` method. For more information, see “`implicitDBTransactionBracketing()`” on page 351.

Is Trace Enabled

Compares the specified trace level with the current trace level of the collaboration.

Inputs

Collaboration The current collaboration object.

Trace level An integer that specifies the trace level to be compared with the current trace level.

Output

Returns `true` if the current system trace level is set to the specified trace level; returns `false` if the two trace levels are not the same.

Notes

The Is Trace Enabled function block is useful in determining if a trace message should or should not be logged. Because tracing can decrease performance, Is Trace Enabled is particularly relevant during the development phase of a project.

Related information

This function block is based on the `BaseCollaboration.isTraceEnabled()` method. For more information, see “`isTraceEnabled()`” on page 352.

JavaException

A constant that represents a problem with the Java code in the collaboration template’s business logic.

Note: This function block is located in the General\APIs\Collaboration Template\Exception\Constants folder.

Output

Returns a String with the value “JavaException”.

ObjectException

A constant that represents an error caused by passing an invalid business object to a function block or by accessing a null object.

Note: This function block is located in the General\APIs\Collaboration Template\Exception\Constants folder.

Output

Returns a String with the value "ObjectException".

OperationException

A constant that represents an error caused by an improperly configured service call that is unable to be sent.

Note: This function block is located in the General\APIs\Collaboration Template\Exception\Constants folder.

Output

Returns a String with the value "OperationException".

Property Exists

Determines whether a specified collaboration configuration property exists.

Inputs

Collaboration The current collaboration object.

Property name

A String that specifies the name of the collaboration configuration property you want to query.

Output

Returns True if the collaboration configuration property exists; otherwise, returns False.

Raise Collaboration Exception

Prepares a collaboration exception to raise it to the next higher level of execution. This function block creates a new exception object with the specified exception type and a message string. Use this form to pass an exception message stored as a string.

Note: This function block is located in the General\APIs\Collaboration Template\Exception folder.

Inputs

Collaboration The current collaboration object.

Exception type

A String that specifies the exception type.

messageNum An integer that specifies the number for the message associated with the exception object.

Notes

The Raise Collaboration Exception function block prepares a collaboration exception to raise to the next higher level of execution. When the collaboration

runtime environment executes the Raise Collaboration Exception function block, it changes the collaboration's execution to the Exception state, then proceeds with the logic of the activity diagram. How the activity diagram responds to the raised exception depends on the termination node of its execution path, as follows:

- If the execution path ends in End Success, control passes to the next higher level of execution.

If this parent diagram's next node is a decision node, the collaboration runtime environment checks for execution branches in this decision node that handle the raised exception. This parent diagram can access the raised exception through the `currentException` system variable.

- If the execution path ends in End Failure, the collaboration runtime environment ends the collaboration, makes an entry in the collaboration's log, and creates an unresolved flow.

The collaboration runtime environment associates with the unresolved flow any exception text that the raised exception contains. If this exception does not contain any exception text, the collaboration runtime environment uses the default message:

Scenario failed.

It is best to explicitly raise an exception when one occurs, rather than to just end in failure. When the code explicitly raises the exception to the collaboration runtime environment, the administrator can use the Flow Manager to view the exception text as part of the unresolved flow. For more information, see "Raising the exception" on page 174.

There is a series of Raise Collaboration Exception function blocks, each of which accomplishes a slightly different task. The Raise Collaboration Exception 1, Raise Collaboration Exception 2, Raise Collaboration Exception 3, Raise Collaboration Exception 4, and Raise Collaboration Exception 5 function blocks enable you to specify up to five message-parameter values for the exception message text. The Raise Collaboration Exception with Parameters function block enables you to specify an array of message-parameter values.

Related information

This function block is based on the `BaseCollaboration.sendEmail()` method. For more information, see "sendEmail()" on page 362.

Raise Collaboration Exception 1

Prepares a collaboration exception to raise to the next higher level of execution. This function block creates a new exception object with the specified exception type and an exception message that is obtained from the collaboration's message file. You identify the message by its message number in the message file. This function block provides the ability to pass a single message-parameter value for the message text.

Note: This function block is located in the `General\APIs\Collaboration Template\Exception` folder.

Inputs

Collaboration The current collaboration object.

Exception type
A String that specifies the exception type.

messageNum An integer that specifies the number for the message associated with the exception object.

Parameter 1 A String that specifies the value for a single message parameter.

Related information

For more information about using the Raise Collaboration Exception function blocks, see “Raise Collaboration Exception” on page 278.

This function block is based on the BaseCollaboration.sendEmail() method. For more information, see “sendEmail()” on page 362.

Raise Collaboration Exception 2

Prepares a collaboration exception to raise to the next higher level of execution. This function block creates a new exception object with the specified exception type and an exception message that is obtained from the collaboration’s message file. You identify the message by its message number in the message file. This function block provides the ability to pass two message-parameter values for the message text.

Note: This function block is located in the General\APIs\Collaboration Template\Exception folder.

Inputs

Collaboration The current collaboration object.

Exception type

A String that specifies the exception type.

messageNum An integer that specifies the number for the message associated with the exception object.

Parameter 1 A String that specifies the value for a single message parameter.

Parameter 2 A String that specifies the value for a single message parameter.

Related information

For more information about using the Raise Collaboration Exception function blocks, see “Raise Collaboration Exception” on page 278.

This function block is based on the BaseCollaboration.sendEmail() method. For more information, see “sendEmail()” on page 362.

Raise Collaboration Exception 3

Prepares a collaboration exception to raise to the next higher level of execution. This function block creates a new exception object with the specified exception type and an exception message that is obtained from the collaboration’s message file. You identify the message by its message number in the message file. This function block provides the ability to pass three message-parameter values for the message text.

Note: This function block is located in the General\APIs\Collaboration Template\Exception folder.

Inputs

- Collaboration** The current collaboration object.
- Exception type**
A String that specifies the exception type.
- messageNum** An integer that specifies the number for the message associated with the exception object.
- Parameter 1** A String that specifies the value for a single message parameter.
- Parameter 2** A String that specifies the value for a single message parameter.
- Parameter 3** A String that specifies the value for a single message parameter.

Related information

For more information about using the Raise Collaboration Exception function blocks, see “Raise Collaboration Exception” on page 278.

This function block is based on the BaseCollaboration.sendEmail() method. For more information, see “sendEmail()” on page 362.

Raise Collaboration Exception 4

Prepares a collaboration exception to raise to the next higher level of execution. This function block creates a new exception object with the specified exception type and an exception message that is obtained from the collaboration’s message file. You identify the message by its message number in the message file. This function block provides the ability to pass four message-parameter values for the message text.

Note: This function block is located in the General\APIs\Collaboration Template\Exception folder.

Inputs

- Collaboration** The current collaboration object.
- Exception type**
A String that specifies the exception type.
- messageNum** An integer that specifies the number for the message associated with the exception object.
- Parameter 1** A String that specifies the value for a single message parameter.
- Parameter 2** A String that specifies the value for a single message parameter.
- Parameter 3** A String that specifies the value for a single message parameter.
- Parameter 4** A String that specifies the value for a single message parameter.

Related information

For more information about using the Raise Collaboration Exception function blocks, see “Raise Collaboration Exception” on page 278.

This function block is based on the BaseCollaboration.sendEmail() method. For more information, see “sendEmail()” on page 362.

Raise Collaboration Exception 5

Prepares a collaboration exception to raise to the next higher level of execution. This function block creates a new exception object with the specified exception type and an exception message that is obtained from the collaboration's message file. You identify the message by its message number in the message file. This function block provides the ability to pass five message-parameter values for the message text.

Note: This function block is located in the General\APIs\Collaboration Template\Exception folder.

Inputs

- Collaboration** The current collaboration object.
- Exception type** A String that specifies the exception type.
- messageNum** An integer that specifies the number for the message associated with the exception object.
- Parameter 1** A String that specifies the value for a single message parameter.
- Parameter 2** A String that specifies the value for a single message parameter.
- Parameter 3** A String that specifies the value for a single message parameter.
- Parameter 4** A String that specifies the value for a single message parameter.
- Parameter 5** A String that specifies the value for a single message parameter.

Related information

For more information about using the Raise Collaboration Exception function blocks, see "Raise Collaboration Exception" on page 278.

This function block is based on the BaseCollaboration.sendEmail() method. For more information, see "sendEmail()" on page 362.

Raise Collaboration Exception with Parameter

Prepares a collaboration exception to raise to the next higher level of execution. This function block provides another way to create a new exception object that contains a specified message in a message file. All parameter values are placed in an array of Objects.

Note: This function block is located in the General\APIs\Collaboration Template\Exception folder.

Inputs

- Collaboration** The current collaboration object.
- Exception type** A String that specifies the exception type.
- messageNum** An integer that specifies the number for the message associated with the exception object.
- Parameters** An array of message-parameter values. Each is sequentially resolved to a parameter in the message text. Within the message

(in the collaboration message file), message parameters are indicated by integers enclosed by braces; for example, {1}.

Notes

This function block is useful in raising an exception object that:

- The collaboration has previously handled. For example, a scenario might get an exception, assign it to a variable, and do some other work.
- Has more than five message parameters. Whereas the other Raise Collaboration Exception function blocks can handle no more than five parameters, the parameter array can contain any number of parameters.

Related information

For more information about using the Raise Collaboration Exception function blocks, see “Raise Collaboration Exception” on page 278.

This function block is based on the BaseCollaboration.raiseException() method. For more information, see “raiseException()” on page 357.

Send Email

Sends an email message asynchronously.

Inputs

Collaboration	The current collaboration object.
Message	A String that contains the text of the email message.
Subject	The subject line of the email message.
Recipients	A Vector that contains email addresses of the message recipients. This Vector contains String objects.

Notes

The Send Email function block can send an email message to the recipients specified in the input Recipients vector if:

- An email address has been specified in the **Email notification address** field in the Collaboration Object Properties dialog.
- The Email collaboration and Email adapter are running. (The Email collaboration is instantiated and configured automatically when InterChange Server starts up and does not require input from the user.) If the Email adapter is not running, Send Email does *not* cause execution of the collaboration to halt.

Note: The Log Error function block automatically sends an error message to an email recipient (assuming that the Email collaboration and adapter are running). The Send Email function block enables you to explicitly send an email message.

Related information

This function block is based on the BaseCollaboration.sendEmail() method. For more information, see “sendEmail()” on page 362.

ServiceCallException

A constant that represents an error caused by a service call failure (for example, if an adapter or application is unavailable).

Note: This function block is located in the General\APIs\Collaboration Template\Exception\Constants folder.

Output

Returns a String with the value "ServiceCallException".

SystemException

A constant that represents an internal error within the InterChange Server system.

Note: This function block is located in the General\APIs\Collaboration Template\Exception\Constants folder.

Output

Returns a String with the value "SystemException".

TransactionException

A constant that represents an error related to the transactional behavior of a transactional collaboration (for example, a rollback failed, or the collaboration could not apply compensation).

Note: This function block is located in the General\APIs\Collaboration Template\Exception\Constants folder.

Output

Returns a String with the value "TransactionException".

Chapter 14. Database connection function blocks

The function blocks in the General\APIs\Database Connection folder provide basic functionality for managing database connections and executing SQL queries in the database. The following sections detail each function block.

Table 71. Summary of database connection function blocks

Function block	Page
Begin Transaction	285
Commit	286
Execute Prepared SQL	287
Execute Prepared SQL with Parameter	288
Execute SQL	288
Execute SQL with Parameter	289
Execute Stored Procedure	290
Get Database Connection	291
Get Database Connection with Transaction	291
Get Next Row	292
Get Update Count	293
Has More Rows	294
In Transaction	294
Is Active	295
Release	295
Roll Back	296

Begin Transaction

Begins an explicit transaction for the current connection.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Exceptions

The Begin Transaction function block can throw the `CwDBConnectionException` exception if a database error occurs.

Notes

The Begin Transaction function block marks the beginning of a new explicit transaction in the current connection. The Begin Transaction, Commit, and Roll Back function blocks together provide management of transaction boundaries for an explicit transaction. This transaction contains SQL queries, which include the SQL statements INSERT, DELETE, or UPDATE, and a stored procedure that includes one of these SQL statements.

Important

Only use Begin Transaction if the connection uses explicit transaction bracketing. If the connection uses implicit transaction bracketing, use of Begin Transaction results in a `CwDBTransactionException` exception.

Before beginning an explicit transaction, you must create a `CwDBConnection` object with the Get Database Connection function block. Make sure that this connection uses explicit transaction bracketing.

Related information

This function block is based on the `CwDBConnection.beginTransaction()` method. For more information, see “`beginTransaction()`” on page 399.

Commit

Commits the active transaction associated with the current connection.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Exceptions

The Commit function block can throw the `CwDBConnectionException` exception if a database error occurs.

Notes

The Commit function block ends the active transaction by committing any changes made to the database associated with the current connection. Together, the Begin Transaction, Commit, and Roll Back function blocks provide management of transaction boundaries for an explicit transaction. This transaction contains SQL queries, which include the SQL statements INSERT, DELETE, or UPDATE, and a stored procedure that includes one of these SQL statements.

Important

Only use Commit if the connection uses explicit transaction bracketing. If the connection uses implicit transaction bracketing, use of Commit results in a `CwDBTransactionException` exception. If you do not end an explicit transaction with Commit (or Roll Back) before the connection is released, InterChange Server implicitly ends the transaction based on the success of the collaboration. If the collaboration is successful, ICS commits this database transaction. If the collaboration is *not* successful, ICS implicitly rolls back the database transaction. Regardless of the success of the collaboration, ICS logs a warning.

Before beginning an explicit transaction, you must create a `CwDBConnection` object with the Get Database Connection function block. Make sure that this connection uses explicit transaction bracketing.

Related information

This function block is based on the `CwDBConnection.commit()` method. For more information, see “`commit()`” on page 400.

Execute Prepared SQL

Executes a prepared SQL query by specifying its syntax.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Query

A String that represents the SQL query to execute in the database.

Exceptions

The Execute Prepared SQL function block can throw the `CwDBSQLException` exception if a database error occurs.

Notes

The SQL statements you can execute include the following (as long as you have the necessary database permissions):

- The SELECT statement to request data from one or more database tables
Use the Has More Rows and Next Row function blocks to access the retrieved data.
- SQL statements that modify data in the database
 - INSERT
 - DELETE
 - UPDATE

The Execute Prepared SQL function block sends the specified *query* string as a prepared SQL statement to the database associated with the current connection. The first time it executes, this query is sent as a string to the database, which compiles the string into an executable form (called a prepared statement), executes the SQL statement, and returns this prepared statement to the function block. The function block then saves this prepared statement in memory. Use Execute Prepared SQL for SQL statements that you need to execute multiple times.

Note: The Execute SQL function block does *not* save the prepared statement and is therefore useful for queries you need to execute only once.

Important

Before executing a query with Execute Prepared SQL, you must obtain a connection to the desired database by generating a `CwDBConnection` object with the Get Database Connection function block.

If the connection uses explicit transaction bracketing, you must explicitly start each transaction with Begin Transaction and end it with either Commit or Roll Back.

- The CALL statement to execute prepared stored procedures with the limitation that this stored procedure *cannot* use any OUT parameters

To execute stored procedures with OUT parameters, use the Execute Stored Procedure function block. For more information, see “Calling stored procedures with executeStoredProcedure()” on page 221.

Related information

This function block is based on the `CwDBConnection.executePreparedSQL()` method. For more information, see “executePreparedSQL()” on page 401.

Execute Prepared SQL with Parameter

Executes a prepared SQL query by specifying its syntax and parameters.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Query

A String that represents the SQL query to execute in the database.

Parameters

A Vector object of arguments to pass to parameters in the SQL query.

Exceptions

The Execute Prepared SQL with Parameter function block can throw the `CwDBSQLException` exception if a database error occurs.

Notes

Refer to the Notes section for the Execute Prepared SQL function block in this chapter.

Related information

This function block is based on the `CwDBConnection.executePreparedSQL()` method. For more information, see “executePreparedSQL()” on page 401.

Execute SQL

Executes a static SQL query by specifying its syntax.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Query

A String that represents the SQL query to execute in the database.

Exceptions

The Execute SQL function block can throw the `CwDBSQLException` exception if a database error occurs.

Notes

The Execute SQL function block sends the specified *query* string as a static SQL statement to the database associated with the current connection. This query is sent as a string to the database, which compiles the string into an executable form and executes the SQL statement, without saving this executable form. Use Execute SQL for SQL statements that you need to execute only once.

Note: The Execute Prepared SQL and Execute Prepared SQL with Parameter function blocks save the executable form (called a prepared statement) and are therefore useful for queries you need to execute multiple times.

Important

Before executing a query with Execute SQL, you must obtain a connection to the desired database by generating a `CwDBConnection` object with the Get Database Connection function block.

The SQL statements you can execute include the following (as long as you have the necessary database permissions):

- The SELECT statement to request data from one or more database tables
Use the Has More Rows and Next Row function blocks to access the retrieved data.
- SQL statements that modify data in the database
 - INSERT
 - DELETE
 - UPDATE

If the connection uses explicit transaction bracketing, you must explicitly start each transaction with the Begin Transaction function block and end it with either the Commit or Roll Back function block.

- The CALL statement to statically execute a stored procedure with the limitation that this stored procedure *cannot* use any OUT parameters
To execute stored procedures with OUT parameters, use the Execute Stored Procedure function block. For more information, see “Calling stored procedures with `executeStoredProcedure()`” on page 221.

Related information

This function block is based on the `CwDBConnection.executeSQL()` method. For more information, see “`executeSQL()`” on page 402.

Execute SQL with Parameter

Executes a static SQL query by specifying its syntax and parameters.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Query

A String that represents the SQL query to execute in the database.

Parameters

A Vector object of arguments to pass to parameters in the SQL query.

Exceptions

The Execute SQL with Parameter function block can throw the `CwDBSQLException` exception if a database error occurs.

Notes

Refer to the Notes section for the Execute SQL function block in this chapter.

Related information

This function block is based on the `CwDBConnection.executeSQL()` method. For more information, see “`executeSQL()`” on page 402.

Execute Stored Procedure

Executes an SQL stored procedure by specifying its name and parameter array.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Stored procedure

A String that represents the stored procedure to execute in the database.

Parameters

A Vector object of arguments to pass to parameters in the SQL query.

Exceptions

The Execute Stored Procedure function block can throw the `CwDBSQLException` exception if a database error occurs.

Notes

The Execute Stored Procedure function block sends a call to the specified *storedProcedure* to the database associated with the current connection. This function block sends the stored-procedure call as a prepared SQL statement; that is, the first time it executes, this stored-procedure call is sent as a string to the database, which compiles the string into an executable form (called a prepared statement), executes the SQL statement, and returns this prepared statement to Execute Stored Procedure. The function block then saves this prepared statement in memory.

Important

Before executing a stored procedure with this function block, you must create a `CwDBConnection` object with the Get Database Connection function block.

To handle any data that the stored procedure returns, use the Has More Rows and Next Row function blocks.

You can also use the Execute SQL, Execute SQL with Parameter, Execute Prepared SQL, or Execute Prepared SQL with Parameter function block to execute a stored procedure as long as this stored procedure does *not* contain OUT parameters. If the stored procedure uses OUT parameters, you *must* use the Execute Stored Procedure function block to execute it.

Unlike the group of Execute SQL and Execute Prepared SQL function blocks, the Execute Stored Procedure function block does not require you to pass in the full SQL statement to execute the stored procedure. With Execute Stored Procedure, you need to pass in only the name of the stored procedure and a Vector parameter array of `CwDBStoredProcedureParam` objects. The Execute Stored Procedure function block can determine the number of parameters from the *storedProcParameters* array and builds the calling statement for the stored procedure.

Related information

This function block is based on the `CwDBConnection.executeStoredProcedure()` method. For more information, see “`executeStoredProcedure()`” on page 404.

Get Database Connection

Establishes a connection to a database.

Inputs

Connection pool name

A String that specifies the name of a valid connection pool.

Output

Returns a `CwDBConnection` object.

Exceptions

The Get Database Connection function block can throw the `CwDBConnectionFactoryException` exception if an error occurs while trying to establish the database connection.

Notes

The Get Database Connection function block obtains a connection from the connection pool that the Connection Pool Name input specifies. This connection provides a way to perform queries and updates to the database associated with the connection. All connections in a particular connection pool are associated with the same database. The function block returns a `CwDBConnection` object through which you can execute queries and manage transactions.

By default, all connections use implicit transaction bracketing as their transaction programming model. To specify a transaction programming model *for a particular connection*, use the Get Database Connection with Transaction function block.

The connection is released when the collaboration object finishes execution. You can explicitly close this connection with the Release function block. You can determine whether a connection has been released with the Is Active function block. For more information, see “Releasing a connection” on page 228.

Related information

This function block is based on the `BaseCollaboration.getDBConnection()` method. For more information, see “`getDBConnection()`” on page 347.

Get Database Connection with Transaction

Establishes a connection to a database using a specific transaction programming model.

Inputs

Connection pool name

A String that specifies the name of a valid connection pool.

Implicit transaction

A boolean value to indicate the transaction programming model to use for the database associated with the connection. Valid values are:

true Database uses implicit transaction bracketing

false Database uses explicit transaction bracketing

Output

Returns a `CwDBConnection` object.

Exceptions

The Get Database Connection With Transaction function block can throw the `CwDBConnectionFactoryException` exception if an error occurs while trying to establish the database connection.

Notes

The Get Database Connection function block obtains a connection from the connection pool that the Connection Pool Name input specifies. This connection provides a way to perform queries and updates to the database associated with the connection. All connections in a particular connection pool as associated with the same database. The function block returns a `CwDBConnection` object through which you can execute queries and manage transactions.

The connection is released when the collaboration object finishes execution. You can explicitly close this connection with the Release function block. You can determine whether a connection has been released with the Is Active function block. For more information, see “Releasing a connection” on page 228.

Related information

This function block is based on the `BaseCollaboration.getDBConnection()` method. For more information, see “`getDBConnection()`” on page 347.

Get Next Row

Gets the next row from a query result.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Output

Returns one row of data from the query result associated with the current connection. Use this method to retrieve results from a query that returns data. Such queries include a SELECT statement and a stored procedure.

Exceptions

The Get Next Row function block can throw the `CwDBSQLException` exception if a database error occurs.

Notes

The Get Next Row function block returns one row of data from the query result associated with the current connection. Use this function block to retrieve results from a query that returns data. Such queries include a SELECT statement and a stored procedure. Only one query can be associated with the connection at a time. Therefore, if you execute another query before Get Next Row returns the last row of data, you lose the query result from the initial query.

Related information

This function block is based on the `CwDBCConnection.nextRow()` method. For more information, see “`nextRow()`” on page 407.

Get Update Count

Determines the number of rows affected by the last write operation to the database.

Inputs

Database connection

A `CwDBCConnection` object that represents the database connection.

Output

Returns an integer that specifies the number of rows affected by the last write operation.

Exceptions

The Get Update Count function block can throw the `CwDBCConnectionException` exception if a database error occurs.

Notes

The Get Update Count function block indicates how many rows have been modified by the most recent update operation in the database associated with the current connection. This function block is useful after you send an UPDATE or INSERT statement to the database and you want to determine the number of rows that the SQL statement has affected.

Important

Before using this method, you must create a `CwDBCConnection` object with the Get Database Connection function block and send a query that updates the database with one of the following function blocks:

- Execute SQL
- Execute SQL with Parameter
- Execute Prepared SQL
- Execute Prepared SQL with Parameter

Related information

This function block is based on the `CwDBCConnection.getUpdateCount()` method. For more information, see “`getUpdateCount()`” on page 405.

Has More Rows

Determines whether the current query result has more rows to process.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Output

Returns true if there are more rows to process in the query result; otherwise, returns false.

Exceptions

The Has More Rows function block can throw the `CwDBSQLException` exception if a database error occurs.

Notes

The Has More Rows function block determines whether the query result associated with the current connection has more rows to be processed. Use this function block to retrieve results from a query that returns data. Such queries include a `SELECT` statement and a stored procedure. Only one query can be associated with the connection at a time. Therefore, if you execute another query before Has More Rows returns false, you lose the data from the initial query.

Related information

This function block is based on the `CwDBConnection.hasMoreRows()` method. For more information, see “`hasMoreRows()`” on page 406.

In Transaction

Determines whether a transaction is in progress in the current database connection.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Output

Returns true if there is a transaction in progress; otherwise, returns false.

Exceptions

The In Transaction function block can throw the `CwDBConnectionException` exception if a database error occurs.

Notes

The In Transaction function block returns a boolean value that indicates whether the current connection has an active transaction; that is, a transaction that has been started but not ended.

Important

Before beginning a transaction, you must create a `CwDBConnection` object with the Get Database Connection function block.

Related information

This function block is based on the `CwDBConnection.inTransaction()` method. For more information, see “`inTransaction()`” on page 406.

Is Active

Determines whether the current connection is active.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Output

Returns true if the current connection is active; otherwise, returns false.

Related information

This function block is based on the `CwDBConnection.isActive()` method. For more information, see “`isActive()`” on page 407.

Release

Releases use of the current connection and returns it to the connection pool.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Exceptions

The Release function block can throw the `CwDBConnectionException` exception.

Notes

The Release function block explicitly releases use of the current connection by the collaboration object. Once released, the connection returns to its connection pool, where it is available for other components (maps or collaborations) that require a connection to the associated database. If you do not explicitly release a connection, the collaboration object implicitly releases it at the end of the current collaboration run. Therefore, you *cannot* save a connection in a static variable and reuse it.

Important

Do *not* use Release if a transaction is currently active. With implicit transaction bracketing, ICS does not end the database transaction until it determines the success or failure of the collaboration. Therefore, use of this function block on a connection that uses implicit transaction bracketing results in a `CwDBTransactionException` exception. If you do not handle this exception explicitly, it also results in an automatic rollback of the active transaction. You can use the In Transaction function block to determine whether a transaction is active.

Related information

This function block is based on the `CwDBConnection.release()` method. For more information, see “release()” on page 408.

Roll Back

Rolls back the active transaction associated with the current connection.

Inputs

Database connection

A `CwDBConnection` object that represents the database connection.

Exceptions

The Roll Back function block can throw the `CwDBTransactionException` exception.

Notes

The Roll Back function block ends the active transaction by rolling back any changes made to the database associated with the current connection. The Begin Transaction, Commit, and Roll Back function blocks together provide management of transaction boundaries for an explicit transaction. This transaction contains SQL queries, which include the SQL statements INSERT, DELETE, or UPDATE, and a stored procedure that includes one of these SQL statements. If the roll back fails, Roll Back throws the `CwDBTransactionException` exception and logs an error.

Important

Only use Roll Back if the connection uses explicit transaction bracketing. If the connection uses implicit transaction bracketing, use of Roll Back results in a `CwDBTransactionException` exception. If you do not end an explicit transaction with Roll Back (or the Commit function block) before the connection is released, InterChange Server Express implicitly ends the transaction based on the success of the collaboration. If the collaboration is successful, InterChange Server Express commits this database transaction. If the collaboration is *not* successful, InterChange Server Express implicitly rolls back the database transaction. Regardless of the success of the collaboration, InterChange Server Express logs a warning.

Related information

This function block is based on the `CwDBConnection.rollback()` method. For more information, see “`rollback()`” on page 409.

Chapter 15. Database stored procedure function blocks

The database stored procedure function blocks provide basic functionality for working with stored procedure parameters. These function blocks are located in the \General\APIs\DB Stored Procedure Param folder.

The following sections detail each function block.

Table 72. Summary of database stored procedure function blocks

Function block	Page
Get Param Type	299
Get Param Value	300
New DB Stored Procedure Param	300

Get Param Type

Retrieves the in/out type of the current stored-procedure parameter.

Inputs

CwDBStoredProcedureParam

The stored-procedure parameter (a CwDBStoredProcedureParam object) whose in/out type you want to retrieve.

Output

Returns the in/out type of the associated stored-procedure parameter as an integer constant.

Notes

The Get Param Type function block returns the in/out parameter type of the current stored-procedure parameter. The in/out parameter type indicates how the stored procedure uses the parameter. The CwDBStoredProcedureParam class represents each in/out type as a constant, as Table 73 shows.

Table 73. Parameter in/out types (Get Param Type function block)

Parameter in/out type	Description	In/out type constant
IN parameter	An IN parameter is <i>input only</i> ; that is, the stored procedure accepts its value as input but does <i>not</i> use the parameter to return a value.	PARAM_IN
OUT parameter	An OUT parameter is <i>output only</i> ; that is, the stored procedure does <i>not</i> read its value as input but does use the parameter to return a value.	PARAM_OUT
INOUT parameter	An INOUT parameter is <i>input and output</i> ; that is, the stored procedure accepts its value as input and also uses the parameter to return a value.	PARAM_INOUT

Related information

This function block is based on the `CwDBStoredProcedureParam.getParamType()` method. For more information, see “`getParamType()`” on page 412.

Get Param Value

Retrieves the value of the current stored-procedure parameter.

Inputs

CwDBStoredProcedureParam

The stored-procedure parameter (a `CwDBStoredProcedureParam` object) whose value you want to retrieve.

Output

Returns the value of the associated stored-procedure parameter as a Java Object

Notes

The Get Param Value function block returns the parameter value as a Java Object (such as `Integer`, `Double`, or `String`). If the value returned to an OUT parameter is the JDBC NULL, Get Param Value returns the `null` constant.

Related information

This function block is based on the `CwDBStoredProcedureParam.getValue()` method. For more information, see “`getValue()`” on page 413.

New DB Stored Procedure Param

Constructs a new instance of `CwDBStoredProcedureParam` that holds argument information for the parameter of a stored procedure.

Inputs

Param Type	The in/out parameter type of the associated stored-procedure parameter.
Param Value	The argument value to send to the stored procedure. This value is one of the following Java data types: <ul style="list-style-type: none">• <code>String</code>• <code>int</code>• <code>Integer</code>• <code>Long</code>• <code>double</code>• <code>Double</code>• <code>float</code>• <code>Float</code>• <code>BigDecimal</code>• <code>boolean</code>• <code>java.sql.Date</code>• <code>java.sql.Time</code>• <code>java.sql.Timestamp</code>

- java.sql.Blob
- java.sql.Clob
- byte[]
- Array
- Struct

Output

Returns a new `CwDBStoredProcedureParam` object to hold the argument information for one argument in the declaration of the stored procedure.

Notes

The New DB Stored Procedure Param function block creates a `CwDBStoredProcedureParam` instance to describe one parameter for a stored procedure. Parameter information includes the following:

- The parameter's in/out type

The function block's first input initializes this in/out parameter type. For a list of valid in/out parameter types, see Table 73 on page 299.

- The parameter value

The function block's second input initializes this parameter value. The `CwDBStoredProcedureParam` class provides one form of its constructor for each of the parameter-value data types it supports. For a list of the mappings between Java data types and JDBC data types for stored-procedure parameters, see Table 62 on page 224.

You provide a Java Vector of stored-procedure parameters to the Execute Stored Procedure function block (in the General\APIs\Database Connection folder), which creates a stored-procedure call from a stored-procedure name and the parameter vector, and sends this call to the database associated with the current connection.

Related information

This function block is based on the `CwDBStoredProcedureParam` constructor. For more information, see "CwDBStoredProcedureParam()" on page 411.

Chapter 16. Exception function blocks

The function blocks in the General\APIs\Collaboration Exception folder provide basic functionality for handling exceptions. The following sections detail each function block.

Table 74. Summary of exception function blocks

Function block	Page
Catch Collaboration Exception	303
Get Message	303
Get Message Number	304
Get Subtype	304
Get Type	305
To String	306

Catch Collaboration Exception

Catches a collaboration exception thrown in the current activity or its subactivities.

Inputs

Collaboration exception

The collaboration exception (a CollaborationException object) the function block is going to catch.

Notes

To define a subactivity, double click the Catch Collaboration Exception function block on the editing canvas.

Get Message

Retrieves the message text from the exception object.

Inputs

Collaboration exception

The collaboration exception (a CollaborationException object).

Output

Returns a String that contains the message text from the exception object.

Notes

The Get Message function block is useful for extracting the exception text from the currentException system variable. This exception text can be included in a call to one of the Raise Collaboration Exception function blocks to ensure that the reason for the exception is raised up to the next higher level of execution.

Note: You can use the To String function block to retrieve the exception type and exception text from the current exception as a formatted string.

Related Information

This function block is based on the `collaborationException.getMessage()` method. For more information, see “`getMessage()`” on page 423.

Get Message Number

Retrieves the message number for the message associated with the exception object.

Inputs

Collaboration exception

The collaboration exception (a `CollaborationException` object).

Output

The integer (int) message number associated with the current exception’s message. If the exception’s message is not from a message file, this function block returns zero (0).

Notes

The Get Message Number function block is useful for obtaining the message number associated with an exception’s message. You can pass this message number to one of the Raise Collaboration Exception function blocks, or to the Log Error function block.

Related Information

This function block is based on the `collaborationException.getMsgNumber()` method. For more information, see “`getMsgNumber()`” on page 424.

Get Subtype

Retrieves the exception subtype from the exception object.

Inputs

Collaboration exception

The collaboration exception (a `CollaborationException` object).

Output

Returns a `String` that contains the exception subtype for the current exception. For more information on valid exception subtypes, see the Notes section.

Notes

For exceptions whose exception type does not adequately identify the cause of the exception, the exception subtype can provide more information. The following exception types most commonly use exception subtypes:

- `JavaException`

The collaboration runtime environment catches Java exceptions and wraps them in a collaboration exception with an associated type of Java exception. A collaboration can use the Get Subtype function block on the collaboration exception to retrieve the original type of the Java exception (that is, the class name of the captured Java exception). However, this is not typically necessary.

- **ServiceCallException**

The `ServiceCallException` exception type occurs if any failure results from a service call. To develop more robust collaborations, you can use the exception subtype to determine the cause of the service-call failure. The valid exception subtypes include:

<code>AppTimeOut</code>	A connector was unable to complete communication with its application.
<code>AppLogOnFailure</code>	A connector was unable to log in to the application.
<code>AppRetrieveByContentFailed</code>	A Retrieve by non-key values, performed on the application, was not able to find any match.
<code>AppMultipleHits</code>	An application found and retrieved more than one entity in response to a Retrieve request.
<code>AppBusObjDoesNotExist</code>	A Retrieve operation was performed on the application, but the entity that the business object represents does not exist in the application database.
<code>AppRequestNotYetSent</code>	In the case of a parallel connector agent, the request was queued up in the agent master but never got dispatched to the application; therefore, you can resend the request. For more information, see “Unsent service call requests” on page 181.
<code>ServiceCallTransportException</code>	There was an error in the transport, and it cannot be determined with certainty whether the request reached the application. For more information, see “Handling runtime transport-related exceptions” on page 179.
<code>AppUnknown</code>	Any type of error that is not one of the other subtypes. If this exception subtype is present, the application operation requested in the service call might be finished or not finished.

Important

The `AppTimeOut`, `AppLogOnFailure`, `AppRetrieveByContent`, `AppMultipleHits`, and `AppUnknown` exception subtypes correspond to outcome-status values that an adapter can return to indicate the cause of failure. Older adapters might not support all of the corresponding outcome-status values. Make sure you rigorously test any adapters that are bound to your collaboration with the Test Connector tool to determine the actual outcome-status values they return.

Related Information

This function block is based on the `collaborationException.getSubType()` method. For more information, see “`getSubType()`” on page 424.

Get Type

Retrieves the collaboration exception type from the exception object. The exception type is a String that identifies the cause of the exception.

Inputs

Collaboration exception

The collaboration exception (a `CollaborationException` object).

Output

Returns a String that contains the exception type for the current exception. Compare this String value with one of the following exception-type static variables:

AnyException	Any type of exception. If there are two exception links, one that tests for a specific type of exception and one that tests for AnyException, the link that tests for the specific type of exception is checked first. If the current exception does not match the specific exception, the link that tests for AnyException is processed next.
AttributeException	Attribute access problem. For example, the collaboration called getDouble() on a String attribute or called getString() on a nonexistent attribute.
JavaException	Problem with Java code in the collaboration logic.
ObjectException	Business object passed to a method was invalid or a null object was accessed.
OperationException	Service call was improperly set up and could not be sent.
ServiceCallException	Service call failed. For example, a connector or application is unavailable.
SystemException	InterChange Server internal error.
TransactionException	Error related to the transactional behavior of a transactional collaboration. For example, rollback failed or the collaboration could not apply compensation.

Notes

The Get Type function block retrieves the exception type from the current exception. The exception type is a String that identifies the cause of the exception.

Related Information

This function block is based on the collaborationException.getType() method. For more information, see “getType()” on page 425.

To String

Formats exception information, including the exception type and text, to a String.

Inputs

Collaboration exception

The collaboration exception (a CollaborationException object).

Output

Returns a String that contains the exception type and text.

Notes

The To String function block formats the exception information for the current exception as follows:

exceptionType: messageText

In the line above, *exceptionType* is the exception object’s exception type and *messageText* is its exception text.

Note: You can use the Get Message function block to retrieve only the exception text from the current exception.

Related Information

This function block is based on the `collaborationException.toString()` method. For more information, see “`toString()`” on page 426.

Chapter 17. Execution function blocks

The function blocks in the General\APIs\Execution Context folder provide execution context functionality. They operate on the global execution context, which is a holder for user-accessible context information that is associated with a given flow. The following sections describe the function blocks in detail.

Table 75. Summary of execution context function blocks

Function block	Page
Get Context	309
MAPCONTEXT	309
New Execution Context	309
Set Context	310

Get Context

Retrieves the specified execution context from the global execution context.

Inputs

Execution context

The global execution context (a CxExecutionContext object).

Context name A String object containing the name of a execution context to obtain from the global execution context.

Output

Returns an instance of the specified execution context.

Related Information

This function block is based on the CwExecutionContext.getContext() method. For more information, see "getContext()" on page 416.

MAPCONTEXT

A String constant used to indicate that the execution context is map-specific.

Output

Returns the MAPCONTEXT String.

New Execution Context

Constructs a new instance of a global execution context.

Output

Returns the new instance of the global execution context.

Notes

The New Execution Context function block returns a global execution context, which is needed to hold the map execution context before calling a map from a collaboration.

Related Information

This function block is based on the `CwExecutionContext()` constructor. For more information, see “`CwExecutionContext()`” on page 415.

Set Context

Sets a particular execution context to be part of the global execution context.

Inputs

- Execution context** The global execution context (a `CwExecutionContext` object).
- Context name** A String object containing the name of a execution context to obtain from the global execution context.
- Context** An object that contains the information for the execution context. For map execution contexts, this object is of type `MapExeContext`.

Related Information

This function block is based on the `CwExecutionContext.setContext()` method. For more information, see “`setContext()`” on page 416.

Chapter 18. Date function blocks

The function blocks in the General\Date folder and its \Formats subfolder provide functionality for working with dates.

Table 76. Summary of date function blocks

Folder	Function block	Page
General\Date	Add Day	311
	Add Month	311
	Add Year	312
	Date After	312
	Date Before	312
	Date Equals	313
	Format Change	313
	Get Day	313
	Get Month	313
	Get Year	314
	Get Year Month Day	314
General\Date\Formats	yyyy-MM-dd	314
	yyyyMMdd	315
	yyyyMMdd HH:mm:ss	315

Add Day

Adds additional days to the original date (as specified by the From date input).

Inputs

- From date** A String object that represents the original date.
- Date format** A String object that represents the date's format.
- Day to add** An integer that specifies the number of days to add to the original date.

Output

Returns a String object that contains the updated date.

Add Month

Adds additional months to the original date.

Inputs

- From date** A String object that represents the original date.
- Date format** A String object that represents the date's format.

Month to add An integer that specifies the number of months to add to the original date.

Output

Returns a String object that contains the updated date.

Add Year

Adds additional years to the original date.

Inputs

From date A String object that represents the original date.

Date format A String object that represents the date's format.

Year to add An integer that specifies the number of years to add to the original date.

Output

A String object that contains the updated date.

Date After

Compares two dates and determines whether Date 1 is after Date 2.

Inputs

Date 1 A String object that represents the first date to compare.

Date 1 format A String object that represents the format of Date 1.

Date 2 A String object that represents the second date in the comparison.

Date 2 format A String object that represents the format of Date 2.

Output

Returns True if Date 1 is after Date 2; otherwise, returns False.

Date Before

Compares two dates and determines whether Date 1 is before Date 2.

Inputs

Date 1 A String object that represents the first date to compare.

Date 1 format A String object that represents the format of Date 1.

Date 2 A String object that represents the second date in the comparison.

Date 2 format A String object that represents the format of Date 2.

Output

Returns True if Date 1 is before Date 2; otherwise, returns False.

Date Equals

Compares two dates and determines whether they are equal.

Inputs

- Date 1** A String object that represents the first date to compare.
- Date 1 format** A String object that represents the format of Date 1.
- Date 2** A String object that represents the second date in the comparison.
- Date 2 format** A String object that represents the format of Date 2.

Output

Returns True if both dates are equal; otherwise, returns False.

Format Change

Changes a date format.

Inputs

- Date** A String object that represents the date you want to reformat.
- Input format** A String object that represents the original format of the date.
- Output format** A String object that represents the new format for the date.

Output

Returns a String object that contains the reformatted date.

Get Day

Returns the numeric day of the month based on date expression.

Inputs

- Date** A String object that represents the date.
- Format** A String object that represents the date's format.

Output

Returns an integer that specifies the day of the month.

Get Month

Returns the numeric month in the year based on date expression.

Inputs

- Date** A String object that represents the date.
- Format** A String object that represents the date's format.

Output

Returns an integer that specifies the numerical value for the month.

Get Year

Returns the numeric year based on date expression.

Inputs

Date A String object that represents the date.
Format A String object that represents the date's format.

Output

Returns an integer that specifies the year.

Get Year Month Day

Extracts the year, month, and day elements from an input date.

Inputs

Date A String object that represents the date.
Format A String object that represents the date's format.

Output

Returns three integers: one that specifies the year, one that specifies the month, and one that specifies the day.

Now

Retrieves today's date.

Inputs

Format A String object that represents the format to be used for the date.

Output

Returns a String object that contains today's date, formatted according to the value given to the Format input.

yyyy-MM-dd

Represents a date format of yyyy-MM-dd (for example, 2003-11-25).

Note: This function block is located in the General\Date\Formats folder.

Output

Returns a String object that contains a date formatted as yyyy-MM-dd.

Notes

This function block does not actually format a date, and it cannot be used as a standalone function block. It must be used in conjunction with one or more of the function blocks in the General\Date folder (for example, with the Format Change or Add Day function block).

yyyyMMdd

Represents a date format of yyyyMMdd (for example, 20031125).

Note: This function block is located in the General\Date\Formats folder.

Output

Returns a String object that contains a date formatted as yyyyMMdd.

Notes

This function block does not actually format a date, and it cannot be used as a standalone function block. It must be used in conjunction with one or more of the function blocks in the General\Date folder (for example, with the Format Change or Add Day function block).

yyyyMMdd HH:mm:ss

Represents a date format of yyyyMMdd HH:mm:ss (for example, 20031125 12:36:40).

Note: This function block is located in the General\Date\Formats folder.

Output

Returns a String object that contains the date formatted as yyyyMMdd HH:mm:ss.

Notes

This function block does not actually format a date, and it cannot be used as a standalone function block. It must be used in conjunction with one or more of the function blocks in the General\Date folder (for example, with the Format Change or Add Day function block).

Chapter 19. Logging and tracing function blocks

The function blocks in the \General\Logging and Tracing folder and its subfolders provide functionality for handling error, informational, warning, and trace messages.

The following sections detail each function block.

Table 77. Summary of the logging and tracing function blocks

Folder	Function block	Page
General\Logging and Tracing	Log error	317
	Log Error ID	317
	Log Information	319
	Log Information ID	319
	Log Warning	320
	Log Warning ID	320
	Trace	321
General\Logging and Tracing\Log Error	Log Error ID 1	318
	Log Error ID 2	318
	Log Error ID 3	318
General\Logging and Tracing\Log Information	Log Information ID 1	319
	Log Information ID 2	319
	Log Information ID 3	320
General\Logging and Tracing\Log Warning	Log Warning ID 1	320
	Log Warning ID 2	321
	Log Warning ID 3	321
General\Logging and Tracing\Trace	Trace ID 1	321
	Trace ID 2	322
	Trace ID 3	322
	Trace on Level	323

Log error

Sends the specified error message to the InterChange Server log file.

Inputs

Message The message to be sent to the log file. This input can be of type Sting, byte, short, int, long, float, or double.

Log Error ID

Sends the error message associated with the specified ID to the InterChange Server log file.

Inputs

ID The ID of the error message you want to log. This input can be of type String, byte, short, int, long, float, or double.

Log Error ID 1

Uses the specified parameter to format the error message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Error folder.

Inputs

ID The ID of the error message you want to log. This input can be of type String, byte, short, int, long, float, or double.

Parameter The parameter used to format the error message. This input can be of type String, byte, short, int, long, float, or double.

Log Error ID 2

Uses the specified two parameters to format the error message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Error folder.

Inputs

ID The ID of the error message you want to log. This input can be of type String, byte, short, int, long, float, or double.

Parameter 1 The first parameter used to format the error message. This input can be of type String, byte, short, int, long, float, or double.

Parameter 2 The second parameter used to format the error message. This input can be of type String, byte, short, int, long, float, or double.

Log Error ID 3

Uses the specified three parameters to format the error message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Error folder.

Inputs

ID The ID of the error message you want to log. This input can be of type String, byte, short, int, long, float, or double.

Parameter 1 The first parameter used to format the error message. This input can be of type String, byte, short, int, long, float, or double.

Parameter 2 The second parameter used to format the error message. This input can be of type String, byte, short, int, long, float, or double.

Parameter 3 The third parameter used to format the error message. This input can be of type String, byte, short, int, long, float, or double.

Log Information

Sends the specified informational message to the InterChange Server log file.

Inputs

Message The message to be sent to the log file. This input can be of type Sting, byte, short, int, long, float, or double.

Log Information ID

Sends the informational message associated with the specified ID to the InterChange Server log file.

Inputs

ID The ID of the informational message you want to log. This input can be of type String, byte, short, int, long, float, or double.

Log Information ID 1

Uses the specified parameter to format the informational message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Information folder.

Inputs

ID The ID of the informational message you want to log. This input can be of type String, byte, short, int, long, float, or double.

Parameter The parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Log Information ID 2

Uses the two specified parameters to format the informational message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Information folder.

Inputs

ID The ID of the informational message you want to log. This input can be of type String, byte, short, int, long, float, or double.

Parameter 1 The first parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Parameter 2 The second parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Log Information ID 3

Uses the three specified parameters to format the informational message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Information folder.

Inputs

ID	The ID of the informational message you want to log. This input can be of type String, byte, short, int, long, float, or double.
Parameter 1	The first parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.
Parameter 2	The second parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.
Parameter 3	The third parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Log Warning

Sends the specified warning message to the InterChange Server log file.

Inputs

Message	The message to be sent to the log file. This input can be of type String, byte, short, int, long, float, or double.
----------------	---

Log Warning ID

Sends the warning message associated with the specified ID to the InterChange Server log file.

Inputs

ID	The ID of the warning message you want to log. This input can be of type String, byte, short, int, long, float, or double.
-----------	--

Log Warning ID 1

Uses the specified parameter to format the warning message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Warning folder.

Inputs

ID	The ID of the warning message you want to log. This input can be of type String, byte, short, int, long, float, or double.
Parameter	The parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Log Warning ID 2

Uses the two specified parameters to format the warning message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Warning folder.

Inputs

ID	The ID of the warning message you want to log. This input can be of type String, byte, short, int, long, float, or double.
Parameter 1	The first parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.
Parameter 2	The second parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Log Warning ID 3

Uses the three specified parameters to format the warning message associated with the ID, and then sends the message to the InterChange Server log file.

Note: This function block is located in the General\Logging and Tracing\Log Warning folder.

Inputs

ID	The ID of the warning message you want to log. This input can be of type String, byte, short, int, long, float, or double.
Parameter 1	The first parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.
Parameter 2	The second parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.
Parameter 3	The third parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Trace

Sends the specified trace message to the InterChange Server log file.

Inputs

Message	The message to be sent to the log file. This input can be of type Sting, byte, short, int, long, float, or double.
----------------	--

Trace ID 1

Uses the specified parameter to format the trace message associated with the ID. Uses the specified level to determine if the trace message is displayed; if tracing in the collaboration is set to the specified level or higher, the trace message is displayed.

Note: This function block is located in the General\Logging and Tracing\Trace folder.

Inputs

ID	The ID of the trace message you want to log. This input can be of type String, byte, short, int, long, float, or double.
Level	The minimum trace level at which the message is going to be displayed. This input can be of type String, byte, short, int, long, float, or double.
Parameter	The parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Trace ID 2

Uses the two specified parameters to format the trace message associated with the ID. Uses the specified level to determine if the trace message is displayed; if tracing in the collaboration is set to the specified level or higher, the trace message is displayed.

Note: This function block is located in the General\Logging and Tracing\Trace folder.

Inputs

ID	The ID of the trace message you want to log. This input can be of type String, byte, short, int, long, float, or double.
Level	The minimum trace level at which the message is going to be displayed. This input can be of type String, byte, short, int, long, float, or double.
Parameter 1	The first parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.
Parameter 2	The second parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

Trace ID 3

Uses the three specified parameters to format the trace message associated with the ID. Uses the specified level to determine if the trace message is displayed; if tracing in the collaboration is set to the specified level or higher, the trace message is displayed.

Note: This function block is located in the General\Logging and Tracing\Trace folder.

Inputs

ID	The ID of the trace message you want to log. This input can be of type String, byte, short, int, long, float, or double.
Level	The minimum trace level at which the message is going to be displayed. This input can be of type String, byte, short, int, long, float, or double.
Parameter 1	The first parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.

- Parameter 2** The second parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.
- Parameter 3** The third parameter used to format the message. This input can be of type String, byte, short, int, long, float, or double.
-

Trace on Level

Displays the trace message if the collaboration's tracing is set to the specified level or higher.

Note: This function block is located in the General\Logging and Tracing\Trace folder.

Inputs

- Message** The message to be displayed. This input can be of type String, byte, short, int, long, float, or double.
- Level** The minimum trace level at which the message is going to be displayed. This input can be of type String, byte, short, int, long, float, or double.

Chapter 20. String function blocks

The function blocks in the General\String folder provide functionality for working with String objects. The following sections describe each function block in detail.

Table 78. Summary of string function blocks

Function block	Page
Append Text	325
If	326
Is Empty	326
Is NULL	326
Left Fill	326
Left String	327
Lower Case	327
Object to String	327
Repeat	327
Replace	328
Right Fill	328
Right String	328
Substring by Position	328
Substring by Value	329
Text Equal	329
Text Equal Ignore Case	329
Text Length	330
Trim Left	330
Trim Right	330
Trim Text	330
Upper Case	330

Append Text

Appends the value of the In String 2 input to the value of the In String 1 input.

Inputs

In String 1 A String object.

In String 2 The String object appended to the string specified in In String 1.

Output

A String object that contains the string from In String 1 with the string from In String 2 appended. For example, if the value of In String 1 is "Hello world." and the value of In String 2 is "How are you?", the output of this function block is the string "Hello world. How are you?".

If

Returns the first value if the condition is true, or the second value if the condition is false.

Inputs

Condition	The condition used to determine the output. This input can be of type boolean or Boolean.
Value 1	A String object.
Value 2	A String object.

Output

Returns the String object associated with Value 1 or Value 2, depending on whether the condition was met.

Is Empty

Returns the second value if the first value is empty.

Inputs

Value 1	A String object.
Value 2	A String object.

Output

Returns the String associated with Value 2 if Value 1 is empty. Otherwise, returns nothing.

Is NULL

Returns the second value if the first value is null.

Inputs

Value 1	A String object.
Value 2	A String object.

Output

Returns the String associated with Value 2 if Value 1 is null. Otherwise, returns nothing.

Left Fill

Returns a String object of the specified length; fills the left with the indicated value.

Inputs

String	A String object.
Fill string	A String object.
Length	An integer that specifies the length of the String object to return.

Output

Returns a filled String object.

Left String

Returns the left portion of the string for the specified number of positions.

Inputs

String The String object you are retrieving.
Length An integer that specifies the number of positions in the String object to return.

Output

Returns a String object that contains the left portion of the original String object.

Lower Case

Changes all characters in a String object to lower case.

Inputs

From string The original String object.

Output

Returns the String object with all lower-case characters.

Object to String

Retrieves a string representation of an object.

Inputs

Object The object to retrieve.

Output

Returns the specified object as a String object.

Repeat

Returns String containing a specified character expression that is repeated a specified number of times.

Inputs

Repeating string The repeating String object you want to retrieve.
Repeat count An integer that specifies the number of times the specified character string must be repeated before it can be retrieved.

Output

Returns the repeated character string.

Replace

Replaces part of a character string with a new character string.

Inputs

String	A String object.
Old string	A String object that contains the particular substring of characters you want to replace.
New string	A String object that contains the new character string you want to use as the replacement.

Output

Returns a String object that contains the updated character string.

Right Fill

Returns a String object of the specified length; fills the right with the indicated value.

Inputs

String	A String object.
Fill string	The String object that is used to fill the right.
Length	An integer that specifies the length of the String object to be returned.

Output

Returns the specified String object, filled with the specified value.

Right String

Returns the right portion of a String object to the specified number of positions.

Inputs

String	The String object to return.
Length	An integer that specifies the number of positions within the character string to return.

Output

Returns a String object with the right portion of the specified string, to the specified number of positions.

Substring by Position

Returns a portion of a String object based on start and end parameters.

Inputs

String	The String object.
---------------	--------------------

- Start position** An integer that specifies the beginning of the portion of the String to be returned.
- End position** An integer that specifies the end of the portion of the String to be returned.

Output

Returns a String object that contains the specified substring.

Substring by Value

Returns a portion of a String object based on specified start and end values. Note that the substring does not include the start and end values, but rather includes everything between those two values.

Inputs

- String** The String object.
- Start value** An integer that specifies the starting value for the substring.
- End value** An integer that specifies the end value for the substring.

Output

Returns a String object that contains the specified substring.

Text Equal

Compares the character strings in two String Objects to determine whether they are equal.

Inputs

- In String 1** The first String object in the comparison.
- In String 2** The second String object in the comparison.

Output

Returns True if the contents of both String objects are equal; otherwise, returns False.

Text Equal Ignore Case

Compares the character strings in two String Objects lexicographically (ignoring case considerations) to determine whether they are equal.

Inputs

- In String 1** The first String object in the comparison.
- In String 2** The second String object in the comparison.

Output

Returns True if the contents of both String objects are equal; otherwise, returns False.

Text Length

Finds the total number of characters in a String object.

Inputs

String The String object.

Output

Returns the length of the String object as one of the following data types: byte, short, int, long, float, or double.

Trim Left

Trims the specified number of characters from the left side of the String object.

Inputs

String The String object you want to trim.

Trim length An integer that specifies the number of characters to trim.

Output

Returns the trimmed String object.

Trim Right

Trims the specified number of characters from the right side of the String object.

Inputs

String The String object you want to trim.

Trim length An integer that specifies the number of characters to trim.

Output

Returns the trimmed String object.

Trim Text

Trims white spaces before and after the characters in a String object.

Inputs

In String The String object you want to trim.

Output

Returns the trimmed String object.

Upper Case

Changes all characters in a String object to upper case.

Inputs

From String The original String object whose characters you want to convert to upper case.

Output

Returns the String object with all characters converted to upper case.

Chapter 21. Utilities function blocks

The function blocks in the General\Utilities folder and its subfolders provide functionality for working with Vector objects, handling exceptions in activities and subactivities, and managing locale issues. The following sections describe each function block in detail.

Table 79. Summary of the utilities function blocks

Folder	Function block	Page
General\Utilities	Catch Error	334
	Catch Error Type	334
	Condition	334
	Loop	337
	Move Attribute in Child	337
	Raise Error	338
	Raise Error Type	338
General\Utilities\Locale	Get Country	335
	Get Language	335
	New Locale	337
	New Locale with Language	338
General\Utilities\Locale\Constants	English	334
	French	334
	German	335
	Italian	336
	Japanese	336
	Korean	336
	Simplified Chinese	338
	Traditional Chinese	339
General\Utilities\Vector	Add Element	333
	Get Element	335
	Iterate Vector	336
	New Vector	338
	Size	339
	To Array	339

Add Element

Adds the specified element to the end of the Vector, increasing its size by one.

Note: This function block is located in the General\Utilities\Vector folder.

Inputs

Vector A java.util.Vector object.

Output

Returns the element.

Catch Error

Catches all the exceptions thrown in the current activity and its subactivities. (You must double-click the function block icon in the editing canvas to define the subactivity.)

Inputs

Error name A String object that specifies the name of the error.

Error message A String object that specifies the contents of the error message.

Catch Error Type

Catches the specified exception type thrown in the current activity and its subactivities. (You must double-click the function block icon in the editing canvas to define the subactivity.)

Inputs

Error type A String object that specifies the type of exception to catch.

Error message A String object that specifies the contents of the error message.

Condition

If the specified condition exists, executes the subactivity defined in "True Action." Otherwise, executes the subactivity defined in "False Action." (You must double-click the function block icon in the editing canvas to define the subactivity.)

Inputs

Condition A boolean that specifies the condition to be met.

English

A constant that represents an English locale.

Note: This function block is located in the General\Utilities\Locale\Constants folder.

Output

An English java.util.Locale object.

French

A constant that represents a French locale.

Note: This function block is located in the General\Utilities\Locale\Constants folder.

Output

A French java.util.Locale object.

German

A constant that represents a German locale.

Note: This function block is located in the General\Utilities\Locale\Constants folder.

Output

A German java.util.Locale object.

Get Country

Determines the country/region code for the current locale.

Note: This function block is located in the General\Utilities\Locale folder.

Inputs

Locale A java.util.Locale object that represents the current locale.

Output

Returns a String object that contains the country/region code for the specified locale. It is typically either an empty string or an uppercase ISO 3166 two-letter code.

Notes

This function block is based on the java.util.Locale.getCountry() method.

Get Element

Gets the element at the specified index in the Vector object.

Note: This function block is located in the General\Utilities\Vector folder.

Inputs

Vector A java.util.Vector object.

Index An integer that specifies the index location.

Output

Returns the element located at the specified index.

Get Language

Determines the language code for the current locale.

Note: This function block is located in the General\Utilities\Locale folder.

Inputs

Locale A java.util.Locale object that represents the current locale.

Output

Returns a String object that contains the language code for the locale. This is either an empty string or a lowercase ISO 639 code.

Notes

This function block is based on the `java.util.Locale.getLanguage()` method.

Italian

A constant that represents an Italian locale.

Note: This function block is located in the `General\Utilities\Locale\Constants` folder.

Output

An Italian `java.util.Locale` object.

Iterate Vector

Iterates through a Vector object.

Note: This function block is located in the `General\Utilities\Vector` folder.

Inputs

Vector A `java.util.Vector` object.

Current index An integer that specifies the index location.

Current element
An object that specifies the element.

Japanese

A constant that represents a Japanese locale.

Note: This function block is located in the `General\Utilities\Locale\Constants` folder.

Output

A Japanese `java.util.Locale` object.

Korean

A constant that represents a Korean locale.

Note: This function block is located in the `General\Utilities\Locale\Constants` folder.

Output

A Korean `java.util.Locale` object.

Loop

Repeats the subactivity until the specified condition is false. (You must double-click the function block icon on the editing canvas to define the subactivity.)

Inputs

Condition A boolean that specifies the condition to be met.

Move Attribute in Child

Moves the value of one attribute to another attribute.

Inputs

Source parent The business object (a BusObj object) that contains the child business object attribute to move.

Source child BO attribute

A String that identifies the name of the child business object that contains the attribute whose value you want to move.

From attribute

A String that identifies the name of the attribute to be moved.

Destination parent

The business object (a BusObj object) to which the original attribute value is to be moved.

Destination child BO attribute

A String that identifies the name of the child business object that contains the attribute whose value you want to replace.

To attribute

A String that identifies the name of the attribute whose value you want to replace with the value of the **From attribute**.

New Locale

Constructs a new locale based on the specified language and county.

Note: This function block is located in the General\Utilities\Locale folder.

Inputs

Language A String object that specifies the current language for the locale.

Country A String object that specifies the current country for the locale.

Output

Returns a java.util.Locale object.

Notes

This function block is based on the util.Locale() method.

New Locale with Language

Constructs a new locale from a language code.

Note: This function block is located in the General\Utilities\Locale folder.

Inputs

Language A String object that specifies the language for the locale.

Output

Returns a new java.util.Locale object.

Notes

This function block is based on the util.Locale() method.

New Vector

Creates a new Vector object.

Note: This function block is located in the General\Utilities\Vector folder.

Output

Returns a new java.util.Vector object.

Raise Error

Throws a new Java exception with the specified message.

Inputs

Message A String object that contains the message for the Java exception.

Raise Error Type

Throws the specified Java exception with the specified message.

Inputs

Error type A String object that identifies the type of Java exception to throw.

Message A String object that contains the message for the Java exception.

Simplified Chinese

A constant that represents a Simplified Chinese locale.

Note: This function block is located in the General\Utilities\Locale\Constants folder.

Output

Returns a Simplified Chinese java.util.Locale object.

Size

Determines the number of elements in a Vector object.

Note: This function block is located in the General\Utilities\Vector folder.

Inputs

Vector A java.util.Vector object.

Output

Returns an integer that specifies the number of elements contained in the Vector object.

To Array

Gets the array representation containing all of the elements in the current Vector object.

Note: This function block is located in the General\Utilities\Vector folder.

Inputs

Vector A java.util.Vector object.

Output

Returns all elements in the Vector object as an array of type Object[].

Traditional Chinese

A constant that represents a Traditional Chinese locale.

Note: This function block is located in the General\Utilities\Locale\Constants folder.

Output

A Traditional Chinese java.util.Locale object.

Part 4. Collaboration API reference

Chapter 22. BaseCollaboration class

The methods documented in this chapter operate on collaboration objects. They are defined on the WebSphere InterChange Server-defined class `BaseCollaboration`. The `BaseCollaboration` class is the base class for all collaborations. All created collaborations are subclasses of `BaseCollaboration`; they all inherit these methods.

Table 80 summarizes the methods of the `BaseCollaboration` class.

Table 80. BaseCollaboration method summary

Method	Description	Page
<code>dropFailedEvent()</code>	Removes a failed event from a collaboration	343
<code>dynamicSend()</code>	Sends a service request to a consumer of the request which is determined dynamically at runtime.	344
<code>existsConfigProperty()</code>	Test the existence of a collaboration configuration property.	345
<code>getConfigProperty()</code>	Retrieve the value of a collaboration configuration property.	346
<code>getConfigPropertyArray()</code>	Retrieve the value of a multi-element collaboration configuration property.	346
<code>getDBConnection()</code>	Establish a connection to a database and returns a <code>CwDBConnection</code> object.	347
<code>getLocale()</code>	Retrieve the locale of the collaboration.	351
<code>getMessage()</code>	Retrieve a message, identified by its message number, from the collaboration message file.	350
<code>getName()</code>	Retrieve the name of this collaboration object.	351
<code>implicitDBTransactionBracketing()</code>	Retrieve the transaction programming model that the collaboration object uses for any connection it obtains.	351
<code>isCallerInRole()</code>	Determine the specific role of a web client.	352
<code>isTraceEnabled()</code>	Compare the specified trace level with the current trace level of the collaboration.	352
<code>logError()</code> , <code>logInfo()</code> , <code>logWarning()</code>	Send an error, information, or warning message to the log file.	353
<code>queryFailedEvents()</code>	Builds a list of failed events that match the query criteria.	355
<code>raiseException()</code>	Raise a collaboration exception.	357
<code>resubmitFailedEvent()</code>	resubmits an event from the saved failed event queue for processing by the collaboration.	360
<code>saveFailedEvent()</code>	Saves a failed event to the saved failed event queue.	362
<code>sendEmail()</code>	Send an email message asynchronously.	362
<code>trace()</code>	Generate a trace message.	363

dropFailedEvent()

This method removes a failed event from a collaboration's event queue. Use this when attempts to resubmit a failed event have been unsuccessful.

Syntax

```
void dropFailedEvent (String ownerName, String connectorname, int index)
```

Parameters

connectorName The name of the connector to which the event belongs.

ownerName The collaboration to which the event belongs.

index The index of the failed event. Get this index using the `queryFailedEvents` method.

Examples

The following example obtains a vector of failed events and then drops them from the queue.

```
// Set up to query the failed events of a specific type for the collaboration.
EventQueryDef queryOption = new EventQueryDef();
// set query condition of busobj name
queryOption.nameBusObj = "TestB0";
// set query condition of connector name
queryOption.nameConnector = "SourceConnector";
// define query condition array
EventQueryDef[] queryOptions = new EventQueryDef[1];
queryOptions[0] = queryOption;
// query failed events with status of
// STATUS_DELIVERY_POSSIBLE_DUPLICATE, no time limitation
FailedEventInfo[] resultArray = queryFailedEvents(queryOptions,
    "", "", FailedEventInfo.STATUS_DELIVERY_POSSIBLE_DUPLICATE);

// get the query result
for (int i = 0; i < resultArray.length; i++) {
    String busObjName = resultArray[i].nameBusObj;
    String nameConnector = resultArray[i].nameConnector;
    int wipIndex = resultArray[i].wipIndex;
    // Drop the failed events with status STATUS_DELIVERY_POSSIBLE_DUPLICATE
    // from the queue.
    dropFailedEvent (busObjName, nameConnector, wipIndex);
}
```

See also

See “`queryFailedEvents()`” on page 355 for information about querying failed events.

dynamicSend()

Sends a service request to a consumer of the request which is determined dynamically at runtime.

Syntax

```
protected void dynamicSend(String consumerType, String consumerName,
    String consumerPort, BusObj busObj, String verb)
    throws CollaborationException
```

Parameters

consumerType
ConsumerType is the type of destination to receive the business object. The value can be either `BaseCollaboration.CONSUMER_CONNECTOR` or `BaseCollaboration.CONSUMER_COLLABORATION`.

consumerName

ConsumerName is the name of the connector or collaboration.

consumerPort

For a collaboration, *consumerPort* is the name of the collaboration's port. If a connector is using **dynamicSend()**, set this parameter to null.

busObj *BusObj* is the name of the business object involved in the service request.

verb *Verb* is the verb name associated with the service request.

Return values

None.

Examples

The following examples shows how to extract information from a business object and then submit a correctly formed dynamic service call.

```
BusObj processingBusObj = new BusObj(triggeringBusObj.getType());
processingBusObj.copy(triggeringBusObj);

/* Extract the dynamic service call parameters from the input BO */
String verb = processingBusObj.getVerb();
String consumerName = processingBusObj.getString("ConsumerName");

/* Consumer type should be a collaboration or a connector only */
String consumerType = processingBusObj.getString("ConsumerType");
String consumerPort = null;

if ((consumerType != null ) &&
(!consumerType.equalsIgnoreCase(BaseCollaboration.CONSUMER_CONNECTOR) )&&
(!consumerType.equalsIgnoreCase(BaseCollaboration.CONSUMER_COLLABORATION)))
{
    // raise an exception here
    raiseException(AnyException, "Invalid consumer type encountered");
}
if
(consumerType.equalsIgnoreCase(BaseCollaboration.CONSUMER_COLLABORATION))
consumerPort = processingBusObj.getString("ConsumerPort");

trace(1, "Making Dynamic Service call with parameters: " +
"ConsumerName = " + consumerName + " ConsumerType = " +
consumerType + " ConsumerPort = " + consumerPort + " Verb = "
+ verb);

//make a dynamic call
dynamicSend(consumerType, consumerName, consumerPort, processingBusObj, verb);
```

Notes

Keep the following in mind when adding dynamic service calls to your collaborations:

- There is no requirement to bind a port at design time. The information needed to process the business object is part of the service call.
- The collaboration can interact with many connectors providing greater flexibility in a dynamic processing environment.
- The collaborations handle the requests synchronously.

existsConfigProperty()

Test the existence of a collaboration configuration property.

Syntax

```
boolean existsConfigProperty(String propertyName)
```

Parameters

propertyName The name of a property defined in the collaboration template.

Return values

Returns true if the property exists; returns false if it does not exist.

Examples

The following example tests for the existence of the VALIDATE_CUSTOMER property.

```
boolean validatePropExists =  
    existsConfigProperty("VALIDATE_CUSTOMER");
```

getConfigProperty()

Retrieve the value of a collaboration configuration property.

Syntax

```
String getConfigProperty(String propertyName)
```

Parameters

propertyName The name of a property defined in the collaboration template.

Return values

Returns the value of the configuration property. If the property does not exist, returns an empty string ("").

Examples

The following example obtains the value of the VALIDATE_CUSTOMER property and assigns it to the variable *validateProp*.

```
String validateProp = getConfigProperty("VALIDATE_CUSTOMER");
```

getConfigPropertyArray()

Retrieve the value of a multi-element collaboration configuration property.

Syntax

```
String[] getConfigPropertyArray(String propertyName)
```

Parameters

propertyName The name of a property defined in the collaboration template.

Return values

An array of property values.

Notes

This method retrieves the value of a multi-element configuration property. A multi-element configuration property consists of a number of values, separated by semicolons.

If the property does not exist, the array is empty. If the property has a single element, the array has only one element.

You can use a multi-element configuration property to get input from a user. The collaboration can then use this multi-element property to build a Retrieve request, in the absence of a business object's key attribute values. The user can specify the attributes that should be used to retrieve the business object by specifying values for the elements of the configuration property.

Examples

The following example retrieves a list of properties associated with the name ATTR_LIST.

```
String[] list = getConfigPropertyArray("ATTR_LIST");
```

getCurrentLoopIndex()

Retrieve the value of the index variable when an iterator is configured as a loop.

Syntax

```
int getCurrentLoopIndex()
```

Parameters

None.

Return values

Returns the value of the loop index variable. Returns zero (0) if it is outside the loop.

Examples

The following example obtains the current loop index:

```
int currentIndex = getCurrentLoopIndex();
```

getDBConnection()

Establish a connection to a database and returns a CwDBConnection object.

Syntax

```
CwDBConnection getDBConnection(String connectionPoolName)  
CwDBConnection getDBConnection(String connectionPoolName,  
boolean implicitTransaction)
```

Parameters

connectionPoolName

The name of a valid connection pool. The method connects to the database whose connection is in this specified connection pool.

implicitTransaction

A boolean value to indicate the transaction programming model to use for the database associated with the connection. Valid values are:

true Database uses implicit transaction bracketing

false Database uses explicit transaction bracketing

Return values

Returns a `CwDBCConnection` object.

Exceptions

`CwDBCConnectionFactoryException` – If an error occurs while trying to establish the database connection.

Notes

The `getDBCConnection()` method obtains a connection from the connection pool that *connectionPoolName* specifies. This connection provides a way to perform queries and updates to the database associated with the connection. All connections in a particular connection pool as associated with the same database. The method returns a `CwDBCConnection` object through which you can execute queries and manage transactions. See the methods in the `CwDBCConnection` class for more information.

By default, all connections use implicit transaction bracketing as their transaction programming model. To specify a transaction programming model *for a particular connection*, provide a boolean value to indicate the desired transaction programming model as the optional *implicitTransaction* argument to the `getDBCConnection()` method. The following `getDBCConnection()` call specifies explicit transaction bracketing for the connection obtained from the `ConnPool` connection pool:

```
conn = getDBCConnection("ConnPool",false);
```

The connection is released when the collaboration object finishes execution. You can explicitly close this connection with the `release()` method. You can determine whether a connection has been released with the `isActive()` method. For more information, see “Releasing a connection” on page 228.

Examples

The following example establishes a connection to the database associated with connections in the `CustConnPool` connection pool. It then uses an implicit transaction to insert and update rows into a table of the database.

```
CwDBCConnection connection = getDBCConnection("CustConnPool");
```

```
// Insert a row  
connection.executeSQL("insert...");
```

```
// Update rows...  
connection.executeSQL("update...");
```

Because the preceding call to `getDBCConnection()` does *not* include the optional second argument, this connection uses implicit transaction bracketing as its transaction programming model (unless the transaction programming model is overridden in the Collaboration Properties dialog of System Manager). Therefore, it does not specify explicit transaction boundaries with `beginTransaction()`,

`commit()`, and `rollback()`. In fact, an attempt to call one of these transaction methods with implicit transaction bracketing generates a `CwDBTransactionException` exception.

Note: You can check the current transaction programming model with the `implicitDBTransactionBracketing()` method.

The following example also establishes a connection to the database associated with connections in the `CustConnPool` connection pool. However, it specifies the use of explicit transaction bracketing for the connection. Therefore, it uses an explicit transaction to contain the inserts and updates on rows in the database tables.

```
CwDBConnection connection = getDBConnection("CustConnPool", false);

// Begin a transaction
connection.beginTransaction();

// Insert a row
connection.executeSQL("insert...");

// Update rows...
connection.executeSQL("update...");

// Commit the transaction
connection.commit();

// Release the connection
connection.release();
```

The preceding call to `getDBConnection()` includes the optional *implicitTransaction* argument to set the transaction programming model to explicit transaction bracketing. Therefore, this examples uses the explicit transaction calls to indicate the boundaries of the transaction. If these transaction methods are omitted, InterChange Server handles the transaction as it would for an implicit transaction.

See also

Chapter 25, "CwDBConnection class", `isActive()`, `release()`

getLocale()

Retrieve the collaboration locale for the current collaboration object.

Syntax

```
java.util.Locale getLocale()
```

Parameters

None.

Return values

A Java Locale object that contains the language and country codes of the collaboration locale. This Locale object must be an instance of the `java.util.Locale` class.

Notes

The `getLocale()` method returns the locale of the current flow. This flow locale is the locale associated with the collaboration object's triggering business object.

Examples

The following example obtains the locale of the collaboration, retrieves the country and language codes from the Locale object, and then reports their values in a trace message:

```
Locale collaborationLocale = getLocale();
String collaborationCountry = collaborationLocale.getCountry();
String collaborationLanguage = collaborationLocale.getLanguage();
trace(3, "THE COUNTRY CODE FOR THE COLLABORATION IS " +
    collaborationCountry +
    ", AND THE LANGUAGE CODE FOR THE COLLABORATION IS " +
    collaborationLanguage +
    ".");
```

getMessage()

Retrieve a message from the collaboration message file.

Syntax

```
public String getMessage(int messageNum)
public String getMessage(int messageNum, Object[] paramArray)
```

Parameters

messageNum

The message number of a message in the collaboration's message file, which is indexed by message number. For information on how to set up a message text file, refer to Chapter 10, "Creating a message file," on page 231.

paramArray

An array of message-parameter values. Each is sequentially resolved to a parameter in the message text. Within the message (in the collaboration message file), message parameters are indicated by integers enclosed by braces; for example, {1}.

Return values

A String object that contains the message text for the message identified by *messageNum*.

Notes

The `getMessage()` method provides two forms:

- The first form takes a message number and retrieves the associated message from the collaboration message file as a String object.
- The second form takes a message number and an array of message-parameter values. It retrieves the associated message from the collaboration message file, replaces its message parameters with the objects in the parameter array, and returns the resulting message as a String object.

For more information on message files and message parameters, see Chapter 10, "Creating a message file," on page 231.

Examples

Suppose your collaboration message file defines the following two messages with message numbers of 8 and 9:

```
8
Error occurred during JDBC URL conversion. Reason:{1}
[EXPL]
An error, indicated by the reason, occurred during the
conversion of a JDBC URL string.
9
Invalid login encountered in command-line arguments. A valid
login must contain a login name and a password.
[EXPL]
A password has been specified but a user name has not. If no
login name is specified, the default login name "crossworlds" is assumed.
```

The following call to `getMessage()` obtains the text associated with message 9:

```
String invalidLogin = getMessage(9);
```

The following call to `getMessage()` obtains the text associated with message 8 and includes a value for the message's Reason parameter:

```
String reason = "Invalid database table.";
Object[] paramList = new Object[1];
paramList[0] = reason;
badConversion = getMessage(8, paramList);
```

The message obtained from the previous `getMessage()` call would be:

```
Error occurred during JDBC URL conversion. Reason:Invalid database table.
```

getName()

Retrieve the name of this collaboration object.

Syntax

```
String getName()
```

Examples

The following example obtains the name of the current collaboration object and logs an informational message:

```
String collabName = getname();
logInfo(collabName + " is starting");
```

implicitDBTransactionBracketing()

Retrieve the transaction programming model that the collaboration object uses for any connection it obtains.

Syntax

```
boolean implicitDBTransactionBracketing()
```

Parameters

None.

Return values

A boolean value to indicate the transaction programming model to be used in all database connections.

Notes

The `implicitDBTransactionBracketing()` method returns a boolean value indicates which transaction programming model the collaboration object assumes is used by *all* connections that it obtains, as follows:

- A value of `true` indicates that all connections use *implicit* transaction bracketing.
- A value of `false` indicates that all connections use *explicit* transaction bracketing.

This method is useful before obtaining a connection to see whether the current transaction programming model is appropriate for that connection.

Note: You can override the transaction programming model for a particular connection with the `getDBConnection()` method.

Examples

The following example ensures that collaboration object uses explicit transaction bracketing for the database associated with the `conn` connection:

```
if (implicitDBTransactionBracketing())
    getDBConnection("ConnPool", false);
```

See also

“Managing the transaction” on page 224

`getDBConnection()`

isCallerInRole()

Returns whether a web client requesting a service is in a specific role. This allows the collaboration to customize the business process based on the client’s identity.

Syntax

```
boolean isCallerInRole(String Role)
```

Parameters

Role The specific role for which you are testing.

Returns

Returns `true` if the client is in the specified *Role*; otherwise returns `false`.

Notes

You must turn on Role-Based Access Control in the Security tab of the System Manager before the collaboration can access the roles. For more details, see the *IBM WebSphere Business Integration Server Administration Guide* for your server.

isTraceEnabled()

Compare the specified trace level with the current trace level of the collaboration.

Syntax

```
public Boolean isTraceEnabled(int traceLevel)
```

Parameters

traceLevel The trace level to compare with the current trace level.

Return values

Returns true if the current system trace level is set to the specified trace level; returns false if the two trace levels are not the same.

Notes

The `isTraceEnabled()` method is useful in determining if a trace message should or should not be logged. Because tracing can decrease performance, this method is useful in the development phase of a project.

Examples

```
if ( isTraceEnabled(3) )
{
    trace("Print this level-3 trace message");
}
```

logError(), logInfo(), logWarning()

Write an error, informational, or warning message to the log destination.

Syntax

```
void logError(String message)
void logError(int messageNum)
void logError(int messageNum, String param [...])
void logError(int messageNum, Object[] paramArray)
void logInfo(String message)
void logInfo(int messageNum)
void logInfo(int messageNum, String param [...])
void logInfo(int messageNum, Object[] paramArray)
void logWarning(String message)
void logWarning(int messageNum)
void logWarning(int messageNum, String param [...])
void logWarning(int messageNum, Object[] paramArray)
```

Parameters

message The message text to be logged.

messageNum The message number of a message in the collaboration's message file, which is indexed by message number. For information on how to set up a message text file, refer to Chapter 10, "Creating a message file," on page 231.

param The value for a single message parameter. There can be up to five message parameters, separated by commas. Each is sequentially resolved to a parameter in the message text.

paramArray An array of message-parameter values. Each is sequentially resolved to a parameter in the message text.

Notes

The `logError()`, `logWarning()`, and `logInfo()` methods send a message to the collaboration's log destination. By default, the log destination is the file `InterchangeSystem.log`. You can change the log destination by entering a value for the `LOG_FILE` parameter in the InterChange Server configuration file, `InterchangeSystem.cfg`. The parameter value can be a file name or `STDOUT`, which writes the log to InterChange Server's command window.

You can also set three other system configuration parameters related to logging. All parameters are located in the InterChange Server configuration file, `InterchangeSystem.cfg`.

- Set the maximum size of the log file with the `MAX_LOG_FILE_SIZE` parameter. Because the default file size is unlimited, you should always set a maximum size.
- Set from one to five archive log files with the `NUMBER_OF_ARCHIVE_LOGS` parameter. The default is five if the parameter is not set.
- Set the `MIRROR_LOG_TO_STDOUT` parameter if you want the error messages to display to `STDOUT` at the same time that they are written to the log file.

For help in deciding whether to use the method that logs an informational, warning, or error message, refer to "Logging messages" on page 193. The message text that appears in the user's log file is prefixed with the word `Info`, `Warning`, or `Error`, depending on the method you use to log the message.

Each of these logging methods has several forms:

- The first form includes all of the text necessary to generate a message. It sends this message to the log destination.
- The second form obtains a message that does *not* have parameters from the collaboration's message file and sends this message to the log destination.
- The third form obtains a message that *does* have parameters from the collaboration's message file. It also provides a list of message-parameter values.
- The fourth form also obtains a message that has parameters from the collaboration's message file. However, it provides the message-parameter values as an array of parameter values.

All forms of the method that take a `messageNum` parameter require the use of a message file that is indexed by message number. For information on how to set up a message text file, refer to Chapter 10, "Creating a message file," on page 231.

In addition to sending a message to the log destination, the `logError()` method also sends the error message to an email recipient if:

- An email address has been specified in the Email notification address field in the Collaboration Object Properties dialog.
- The Email collaboration and Email connector are running. (The Email collaboration is instantiated and configured automatically when InterChange Server starts up and does not require input from the user.)

Notes:

1. The CollaborationFoundation provides the `SEND_EMAIL` configuration property to allow you to specify by message number which messages will generate email notification. For more information, see "CollaborationFoundation template" on page 29.

2. The `logError()` method automatically sends an error message to an email recipient (assuming that the Email collaboration and Email adapter are running). The `sendEmail()` method allows you to explicitly send an email message.

Examples

The following example logs an error message, using `getString()` to obtain an attribute value in the message.

```
logError("Incorrect customer: CustomerID: "
    + fromCustomerBusObj.getString("CustomerID"));
```

The following example logs an error message whose text is contained in the collaboration's message file. The message, which is number 10 in the message file, takes two parameters: customer last name (LName attribute) and customer first name (FName attribute).

```
logError(10, customer.get("LName"), customer.get("FName"));
```

The following example logs an error message using an array of parameters. For the purpose of illustration, the example uses an array with just two parameters. The example declares the array `args`, which has two elements, the customer ID and the customer name. The `logError()` method then logs an error, using message number 12 and the values in the `args` array.

```
Object[] args =
{
    fromCustomerBusObj.getString("CustomerID"),
    fromCustomerBusObj.getString("CustomerName");
}
logError(12, args);
```

queryFailedEvents()

This method queries the failed event database to locate any events that meet the selection criteria.

Syntax

```
queryFailedEvents (EventQueryDef[] queryOptions, String startDateTime,
    String endDateTime, int eventStatus, int maxrows)
```

Parameters

<i>queryOptions</i>	An array of selection criteria with the following format: <i>nameConnector</i> Name of the connector of interest. <i>nameCollaboration</i> Name of collaboration of interest. <i>nameBusObj</i> Name of the business object of interest. <i>verb</i> name of the verb of interest. If left null, all verbs will be selected.
<i>startDateTime</i>	The earliest date and time at which the event occurred. The format of the time is MONTH DD YYYY HH24:MI:SS.
<i>endDateTime</i>	The latest date and time at which the event occurred. The format of the time is MONTH DD YYYY HH24:MI:SS.

Note: HH24 indicates hours in 24-hour format.

eventStatus An integer that indicates the status of the event which can be one of the following:

- STATUS_ANY_UNRESOLVED
- STATUS_FAILED
- STATUS_DEFERRED_RECOVERY
- STATUS_SERVICECALL_IN_TRANSIT
- STATUS_DELIVER_POSSIBLE_DUPLICATE
- STATUS_SERVICECALL_WAITING
- STATUS_USER_GENERATED

maxrows Maxrows limits the number of elements returned by the method. If this parameter is omitted or set to zero, there is no limit to the number of elements returned.

The results returned is dependant on both the *eventStatus* and how you specify the time. Table 81 shows the what various time parameters will return in the results.

Table 81. Results for *queryFailedEvents* based on time parameters

startDateTime	endDateTime	Results
Not null	Not null	All events that meet the <i>eventStatus</i> that occurred between <i>startDateTime</i> and <i>endDateTime</i>
Null	Not null	All events that meet the <i>eventStatus</i> that occurred before <i>endDateTime</i>
Not null	Null	All events that meet the <i>eventStatus</i> that occurred after <i>startDateTime</i>
Not null	Not null	All events that meet the <i>eventStatus</i> regardless of time.

Returns

An array of failed events, **FailedEventInfo**, each element having the following structure:

nameOwner *NameOwner* is the name of the collaboration to which this event belongs.

nameConnector *NameConnector* is the name of the connector to which this event belongs.

nameBusObj *NameBusObj* is the name of the business object involved in this event.

nameVerb *NameVerb* is the verb executing during this event.

strTime *StrTime* is the time of the event.

strMessage *strMessage* is a message associated with the failing event.

wipIndex *wipIndex* is an integer that indicates the event's position.

EventKeyAttrDef[]

An array that consists of *nkeys* elements of pairs of event attributes. The array has the following structure:

keyName

KeyName is the attribute name.

	<i>keyValue</i>	<i>Key</i> Value is the attribute value.
<i>nkeys</i>	<i>NKeys</i> is the number of key attributes in the EventKeyAttrDef array.	
<i>eventStatus</i>	<i>EventStatus</i> is an integer that indicates the status of the event which can be one of the following: STATUS_ANY_UNRESOLVED STATUS_FAILED STATUS_DEFERRED_RECOVERY STATUS_SERVICECALL_IN_TRANSIT STATUS_DELIVER_POSSIBLE_DUPLICATE STATUS_SERVICECALL_WAITING STATUS_USER_GENERATED	
<i>expirationTime</i>	<i>ExpirationTime</i> is the time at which the event expires for a Long-Lived Business Process (LLBP).	
<i>scenarioName</i>	<i>ScenarioName</i> is the name of the scenario occurring at the time of the event.	
<i>scenarioState</i>	<i>ScenarioState</i> is the completion state of the running scenario.	

Examples

The following example queries for all events that match the status STATUS_USER_GENERATED.

```
// Set up to query the failed events of a specific type for the collaboration.
EventQueryDef queryOption = new EventQueryDef();
// set query condition of busobj name
queryOption.nameBusObj = "TestB0";
// set query condition of connector name
queryOption.nameConnector = "SourceConnector";
// define query condition array
EventQueryDef[] queryOptions = new EventQueryDef[1];
queryOptions[0] = queryOption;
// query failed events with status of user_generated, no time
// limitation
FailedEventInfo[] resultArray = queryFailedEvents(queryOptions,
    "", "", FailedEventInfo.STATUS_USER_GENERATED);

// get the query result
for (int i = 0; i < resultArray.length; i++) {
    String busObjName = resultArray[i].nameBusObj;
    logInfo(busObjName + " failed");
}
```

See also

- “dropFailedEvent()” on page 343
- “resubmitFailedEvent()” on page 360
- “saveFailedEvent()” on page 362

raiseException()

Prepare a collaboration exception to raise to the next higher level of execution.

Syntax

```
void raiseException(String exceptionType, String message)
void raiseException(String exceptionType, int messageNum,
    String parameter[,...])
void raiseException (String exceptionType, int messageNum,
    Object[] paramArray)
void raiseException(CollaborationException exceptionObject)
```

Parameters

<i>exceptionType</i>	The exception type for the exception to be raised. Specify this exception type as one of the following exception-type static variables, which identify the cause of the collaboration exception:
AnyException	Any type of exception
AttributeException	Attribute access problem. For example, the collaboration called <code>getDouble()</code> on a <code>String</code> attribute or called <code>getString()</code> on a nonexistent attribute.
JavaException	Problem with Java code in collaboration logic.
ObjectException	Business object passed to a method was invalid or a null object was accessed.
OperationException	Service call was improperly set up and could not be sent.
ServiceCallException	Service call failed. For example, a connector or application is unavailable.
SystemException	Any internal error within the InterChange Server system.
TransactionException	Error related to the transactional behavior of a transactional collaboration. For example, rollback failed or the collaboration could not apply compensation.
<i>message</i>	A text string that contains the exception message.
<i>messageNum</i>	The message number of a message in the collaboration's message file, which is indexed by message number. For information on how to set up a message text file, refer to Chapter 10, "Creating a message file," on page 231.
<i>parameter</i>	A value for a single message parameter. There can be up to five message parameters, separated by commas. Each is sequentially resolved to a parameter in the message text.
<i>paramArray</i>	An array of message-parameter values. Each is sequentially resolved to a parameter in the message text.
<i>exceptionObject</i>	The name of a <code>CollaborationException</code> exception-object variable.

The following explanations and code examples need to be provided in the Collaboration API:

Notes

The `raiseException()` method prepares a collaboration exception to raise to the next higher level of execution. When the collaboration runtime environment executes the `raiseException()` call, it changes the collaboration's execution to the Exception state, then proceeds with the logic of the activity diagram. How the activity diagram responds to the raised exception depends on the termination node of its execution path, as follows:

- If the execution path ends in End Success, control passes to the next higher level of execution.

If this parent diagram's next node is a decision node, the collaboration runtime environment checks for execution branches in this decision node that handle the raised exception. This parent diagram can access the raised exception through the `currentException` system variable.

- If the execution path ends in End Failure, the collaboration runtime environment ends the collaboration, makes an entry in the collaboration's log, and creates an unresolved flow.

The collaboration runtime environment associates with the unresolved flow any exception text that the raised exception contains. If this exception does not contain any exception text, the collaboration runtime environment uses the default message:

Scenario failed.

It is best to explicitly raise an exception when one occurs, rather than to just end in failure. When the code explicitly raises the exception to the collaboration runtime environment, the administrator can use the Flow Manager to view the exception text as part of the unresolved flow. For more information, see "Raising the exception" on page 174.

The `raiseException()` method has several forms:

- The *first* form creates a new exception object with the specified exception type and a message string. Use this form to pass an exception message stored as a string. You might also send this string message to one of the log methods to log it.
- The *second* form creates a new exception object with the specified exception type and an exception message that is obtained from the collaboration's message file. You identify the message by its message number in the message file. This form of the method call provides the ability to pass up to five message-parameter values for the message text. Separate these message parameters with commas. In the message text (within the message file), parameters are specified by a number within curly braces, such as {1}. The `raiseException()` method should provide a value for each message parameter in the message.
- The *third* form provides another way to create a new exception object that contains a specified message in a message file. Its message-parameter array of Objects behaves similarly to the String parameter list in the second form of this method. However, whereas each message-parameter value in the String list is specified separately, this form places all parameter values in an array of Objects. This form is useful in raising an exception object that:
 - The collaboration has previously handled. For example, a scenario might get an exception, assign it to a variable, and do some other work.
 - Has more than five message parameters. Whereas the String list can contain no more than five parameters, the parameter array can contain any number of parameters.

- The *fourth* form does *not* create an exception. Instead, it just raises the specified exception object (CollaborationException object) that is provided as an argument.

Note: All forms of the method that take a *messageNum* parameter require the use of a message file indexed by message number. For information on how to set up a message text file, refer to Chapter 10, “Creating a message file,” on page 231.

Examples

This section provides examples of each of the forms of the `raiseException()` method:

1. The following example uses the first form of the method to raise an exception of `ServiceCallException` type. The text is passed directly into the method call.

```
raiseException(ServiceCallException, "Attempt to validate
Customer failed.");
```

2. The next example uses the second form of the method to raise an exception of `OperationException` type, whose message appears in the message file is as follows:

```
23
Customer update failed for CustomerID={1} CustomerName={2}
```

This `raiseException()` call retrieves message 23 and retrieves the values of the message’s two parameters (customer ID and name) from the `fromCustomer` variable to generate the exception message:

```
raiseException(OperationException, 23,
    fromCustomer.getString("CustomerID"),
    fromCustomer.getString("CustomerName"));
```

3. The following example uses the third form to send message-parameter values as an array of `Objects`.

For example, assume the message file includes the following message text:

```
2000
Collaboration Message: BOName: {1} with Verb: {2} encountered
an undefined error.
```

The following code creates a parameter array of `Objects`, loads values into it, and calls the `raiseException()` method:

```
Object[] myParamArray = new Object[2];

myParamArray[0] = triggeringBusObj.getType();
myParamArray[1] = triggeringBusObj.getVerb();

raiseException(AnyException, 2000, myParamArray);
```

4. The final example uses the fourth form of the method to raise a previously handled exception. The system-defined variable `currentException` is the exception object that contains the exception.

```
raiseException(currentException);
```

resubmitFailedEvent()

This method resubmits a failed event to a collaboration for processing.

Syntax

```
resubmitFailedEvent (String ownerName, String connectorName, int index,
    int option, int flowTraceLevel)
```


Parameters

<i>ownerName</i>	<i>OwnerName</i> is the name of the collaboration to which you are submitting the failed event.
<i>connectorName</i>	<i>ConnectorName</i> is the name of the connector to which you are submitting the failed event.
<i>index</i>	<i>Index</i> is the index of the failed event.
<i>option</i>	<i>Option</i> specifies how to resubmit the failed event. <i>Option</i> can have one of the following values: FailedEventInfo.RESUBMIT_ORIGINAL_BUSOBJ The method is to resubmit the original business object that failed. FailedEventInfo.RESUBMIT_NEW_BUSOBJ The method is to request a new business object from the connector and submits the new business object for processing.
<i>flowtracelevel</i>	<i>flowtracelevel</i> is an integer that indicates the trace level for the resubmitted failed flow.

Examples

The following example resubmits business objects with event status STATUS_SERVICECALL_IN_TRANSIT for reprocessing.

```
// Set up to query the failed events of a specific type for the collaboration.
EventQueryDef queryOption = new EventQueryDef();
// set query condition of busobj name
queryOption.nameBusObj = "TestB0";
// set query condition of connector name
queryOption.nameConnector = "SourceConnector";
// define query condition array
EventQueryDef[] queryOptions = new EventQueryDef[1];
queryOptions[0] = queryOption;
// query failed events with status of STATUS_SERVICECALL_IN_TRANSIT,
// no time limitation
FailedEventInfo[] resultArray = queryFailedEvents(queryOptions,
    "", "", FailedEventInfo.STATUS_SERVICECALL_IN_TRANSIT);
int newStatus = FailedEventInfo.RESUBMIT_ORIGINAL_BUSOBJ;
int myFlowTraceLevel = 3;

// get the query result
for (int i = 0; i < resultArray.length; i++) {
    String busObjName = resultArray[i].nameBusObj;
    String nameConnector = resultArray[i].nameConnector;
    int wipIndex = resultArray[i].wipIndex;

// Resubmit the failed events with status STATUS_SERVICECALL_IN_TRANSIT
// from queue.
    resubmitFailedEvent (busObjName, nameConnector, wipIndex, newStatus,
        myFlowTraceLevel);
}
```

See also

“queryFailedEvents()” on page 355

“saveFailedEvent()” on page 362

saveFailedEvent()

This method saves a failed event and sets its status to STATUS_USER_GENERATED.

Syntax

```
saveFailedEvent (BusObj busObj)
```

Parameters

busObj *BusObj* is the business object to be saved. The saved business object has an identical structure to the original failing event and your structure should be defined before calling the method. The eventStatus of the failing event becomes STATUS_USER_GENERATED.

Examples

```
// Synchronize product information in a batch using wrapper business
// object ProductSet.
// Create an empty parent bo
BusObj parentBo = new BusObj("ProductSet");

// Set the values which are the same as the triggeringBusObj
parentBo.set("ProductSetId", triggeringBusObj.get("ProductSetId"));
parentBo.set("CreateDate", triggeringBusObj.get("CreateDate"));

// Set Verb to the parent bo
parentBo.setVerb(triggeringBusObj.getVerb());

// Add the failed business object to parent bo. Here iterProduct
// is the child BO which gets failed during iteration process.
parentBo.set("Products", iterProduct);

// Call collaboration API to save the failed event
saveFailedEvent(parentBo);
```

Notes

- The saved business object must be supported by the collaboration.
- In a collaboration group, saving failed events can only be done at the head of the collaboration group.
- This method does not support the Access Framework.

See also

“queryFailedEvents()” on page 355

sendEmail()

Send an email message asynchronously.

Syntax

```
void sendEmail(String message, String subject, Vector recipients)
```

Parameters

message The text of the email message.

<i>subject</i>	The subject line of the email message
<i>recipients</i>	A Vector that contains email addresses of the message recipients. This vector contains String objects; multiple objects are separated by commas.

Notes

The `sendEmail()` method can send an email message to the recipients specified in the *recipients* vector if:

- An email address has been specified in the Email notification address field in the Collaboration Object Properties dialog.
- The Email collaboration and Email adapter are running. (The Email collaboration is instantiated and configured automatically when InterChange Server starts up and does not require input from the user.) If the Email adapter is not running, `sendEmail()` does *not* cause execution of the collaboration to halt.

Note: The `logError()` method automatically sends an error message to an email recipient (assuming that the Email collaboration and adapter are running). The `sendEmail()` method allows you to explicitly send an email message.

Examples

```
// Initialize the Vector for the list of email addresses
Vector emailList = new Vector();

// Add as many email addresses as Strings to the Vector
emailList.add("dbadmin@us.ibm.com, netadmin@us.ibm.com,
             cwadmin@us.ibm.com");

// Initialize the message and subject as Strings
String message = "This is the body of the email";
String subject = "This is the subject of the email";

// Make the call to sendEmail()
sendEmail(message, subject, emailList);
```

trace()

Write a trace message to the log destination.

Syntax

```
void trace(String traceMsg)
void trace(int traceLevel, String traceMsg)
void trace(int traceLevel, int messageNum)
void trace(int traceLevel, int messageNum, String param [...])
void trace(int traceLevel, int messageNum, Object[] paramArray)
```

Parameters

<i>traceLevel</i>	The tracing level that is used to determine which trace messages are output. The method writes the trace message when the trace level for the collaboration object is greater than or equal to this <i>traceLevel</i> value. You should define the trace levels for this collaboration and document them so that the administrator knows which level to use for the collaboration object.
<i>traceMsg</i>	The trace-message text that is written to the trace file.
<i>messageNum</i>	The message number of a message in the collaboration's message

file, which is indexed by message number. For information on how to set up a message text file, refer to Chapter 10, “Creating a message file,” on page 231.

<i>param</i>	A value for a single message parameter. There can be up to five message parameters, separated by commas. Each is sequentially resolved to a parameter in the message text.
<i>paramArray</i>	An array of message-parameter values. Each is sequentially resolved to a parameter in the message text.

Notes

The `trace()` method sends a trace message to the collaboration’s log destination. By default, the log destination is the file `InterchangeSystem.log`. You can change the log destination by entering a value for the `LOG_FILE` parameter in the InterChange Server configuration file, `InterchangeSystem.cfg`. The parameter value can be a file name or `STDOUT`, which writes the log to InterChange Server’s command window.

You can also set three other system configuration parameters related to trace logging. All parameters are located in the configuration file, `InterchangeSystem.cfg`:

- Set the maximum size of the trace file with the `MAX_TRACE_FILE_SIZE` parameter. Because the default file size is unlimited, you should always set a maximum size.
- Set from one to five archive trace files with the `NUMBER_OF_ARCHIVE_TRACES` parameter. The default is five if the parameter is not set.
- Set the `MIRROR_TRACE_TO_STDOUT` parameter if you want the error messages to display to `STDOUT` at the same time that they are written to the trace file. The default value is `false`; messages are not simultaneously written to `STDOUT`.

The `trace()` method has several forms:

- The first form of the method takes just a string message that appears when tracing is set to level 1 or above.
- The second form takes a trace level and a string message that appears when tracing is set to the specified level or a higher level.
- The third form takes a trace level and a number that represents a message in the collaboration’s message file. The entire message text appears in the message file and is printed as it is, without parameters, when tracing is set to the specified level or a higher level.
- The fourth form takes a trace level, a number that represents a message in the collaboration’s message file, and one or more parameters to be used in the message. You can send up to five parameter values to be used with the message by separating the values with commas.
- The fifth form takes a trace level, a number that represents a message in the collaboration’s message file, and an array of parameter values.

A collaboration object can be configured to generate a system-generated trace or a collaboration-generated trace. The `trace()` method generates a message that the collaboration object prints if configured to print a collaboration-generated trace. For help in deciding when to use tracing, refer to “Adding trace messages” on page 195.

Examples

The following example uses the second form of the method to generate a Level 2 trace message with the supplied text of the message:

```
trace (2, "Starting to trace at Level 2");
```

The following example uses the fourth form of the method to write message 201 in the collaboration's message file, if the collaboration object trace level is 2 or higher. The message has two parameters, a name and a year, for which this method call passes values.

```
trace(2, 201, "DAVID", "1961");
```

Chapter 23. BusObj class

The methods documented in this chapter operate on objects of the BusObj class. These objects represent WebSphere InterChange Server business objects.

Note: The BusObj class is used for both collaboration development and mapping; check the Notes section for each method's usage issues.

Table 82 lists the methods of the BusObj class.

Table 82. BusObj method summary

Method	Description	Page
copy()	Copy all attributes values from the input business object to this one.	368
duplicate()	Create a business object (BusObj object) exactly like this one.	369
equalKeys()	Compare this business object's key attribute values with those in the input business object.	369
equals()	Compare this business object's attribute values with those in the input business object, including child business objects.	370
equalsShallow()	Compare this business object's attribute values with those in the input business object, excluding child business objects from the comparison.	371
exists()	Check for the existence of a business object attribute with a specified name.	372
getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString()	Retrieve the value of a single attribute from a business object.	372
getLocale()	Retrieves the locale of the business object's data.	374
getType()	Retrieve the name of the business object definition on which this business object was based.	375
getVerb()	Retrieve this business object's verb.	375
isBlank()	Find out whether the value of an attribute is set to a zero-length string.	376
isKey()	Find out whether a business object's attribute is defined as a key attribute.	376
isNull()	Find out whether the value of a business object's attribute is null.	377
isRequired()	Find out whether a business object's attribute is defined as a required attribute.	378
keysToString()	Retrieve the values of a business object's primary key attributes as a string.	378
set()	Set a business object's attribute to a specified value of a particular data type.	379
setDefaultAttrValues()	Set all attributes to their default values.	380

Table 82. BusObj method summary (continued)

Method	Description	Page
setKeys()	Set the values of this business object's key attributes to the values of the key attributes in another business object.	380
setVerb()	Set the verb of a business object.	381
setWithCreate()	Set a business object's attribute to a value of a specified data type.	382
toString()	Return the values of all attributes in a business object as a string.	382
validData()	Checks whether a specified value is a valid data type for a specified attribute.	383

copy()

Copy all attributes values from the input business object to this one.

Syntax

```
void copy(BusObj inputBusObj)
```

Parameters

inputBusObj The name of the business object whose attributes values are copied into the current business object.

Notes

The `copy()` method copies the entire business object, including all child business objects and child business object arrays. This method does not set a reference to the copied object. Instead, it clones all attributes; that is, it creates separate copies of the attributes.

Examples

The following example copies the values contained in `sourceCustomer` to `destCustomer`.

```
destCustomer.copy(sourceCustomer);
```

The following example creates three business objects (`myBusObj`, `myBusObj2`, and `mySettingBusObj`) and sets the `attr1` attribute of `myBusObj` with the value in `mySettingBusObj`. It then clones all attributes of `myBusObj` to `myBusObj2`.

```
BusObj myBusObj = new BusObj();
BusObj myBusObj2 = new BusObj();

BusObj mySettingBusObj = new BusObj();

myBusObj.set("attr1", mySettingBusObj);
myBusObj2.copy(myBusObj);
```

After this code fragment executes, `myBusObj.attr1` and `myBusObj2.attr1` are *both* set to the `mySettingBusObj` business object. However, if `mySettingBusObj` is changed in any way, `myBusObj.attr1` changes but `myBusObj2.attr1` does not. Because the attributes of `myBusObj2` were set with `copy()`, their values were cloned. Therefore, the value of `attr1` in `myBusObj2` is still the original `mySettingBusObj.attr1` value *before* the change.

duplicate()

Create a business object (BusObj object) exactly like this one.

Syntax

```
BusObj duplicate()
```

Return values

The duplicate business object.

Exceptions

CollaborationException—The duplicate() method can set the following exception type for this exception: ObjectException.

Notes

This method makes a clone of the business object and returns it. You must explicitly assign the return value of this method call to a declared variable of BusObj type.

Examples

The following example duplicates sourceCustomer in order to create destCustomer.

```
BusObj destCustomer = sourceCustomer.duplicate();
```

equalKeys()

Compare this business object's key attribute values with those in the input business object.

Syntax

```
boolean equalKeys(BusObj inputBusObj)
```

Parameters

inputBusObj A business object to compare with this business object.

Return values

Returns true if the values of all key attributes are the same; returns false if they are not the same.

Exceptions

CollaborationException—The equalKeys() method can set the following exception type for this exception:

- ObjectException – Set if the business object argument is invalid.

See also

equals(), equalsShallow()

Notes

This method performs a shallow comparison; that is, it does not compare the keys in child business objects.

Examples

The following example compares the key values of order2 to those in order1.

```
boolean areEqual = order1.equalKeys(order2);
```

equals()

Compare this business object's attribute values with those in the input business object, including child business objects.

Syntax

```
-boolean equals(Object inputBusObj)
```

Parameters

inputBusObj A business object to compare with this business object.

Return values

Returns true if the values of all attributes are the same; otherwise, returns false.

Exceptions

CollaborationException—The equals() method can set the following exception type for this exception:

- **ObjectException**—Set if the business object argument is invalid.

Notes

This method compares this business object's attribute values with those in the input business object. If the business objects are hierarchical, the comparison includes all attributes in the child business objects.

1. Passing in the business object as an Object ensures that this equals() method overrides the Object.equals() method.
2. In the comparison, a null value in the source object is considered equivalent to any value to which it is compared and results in a return of true. The results from this method may not be symmetrical based on the order of comparison. A comparison of A.equals(B) may not be the same as B.equals(A). The second example shows the results of various comparisons of the same objects.
3. The order of the children does effect the results of this method. If everything else matches, a mismatch in the order will return a false value.
4. The verb has no effect on the outcome of the comparison.

See also

equalKeys(), equalsShallow()

Examples

The following example compares all attributes of order2 to all attributes of order1 and assigns the result of the comparison to the variable areEqual. The comparison includes the attributes of child business objects, if any.

```
boolean areEqual = order1.equals(order2);
```

The next example compares triggeringBusObj to sourceBusObj and shows how the order of the comparison affects the results.

```

logInfo("***Comparing triggering B0 to empty Source B0");
if (triggeringBusObj.equals(SourceBusObj)) {
    logInfo("**** triggeringBusObj.equals(SourceBusObj) - TRUE");
} else
    logInfo("**** triggeringBusObj.equals(SourceBusObj) - FALSE");
if (SourceBusObj.equals(triggeringBusObj)) {
    logInfo("**** SourceBusObj.equals(triggeringBusObj) - TRUE");
} else
    logInfo("**** SourceBusObj.equals(triggeringBusObj) - FALSE");

logInfo("*** Copying Source B0 verb from triggering B0");
SourceBusObj.copy(triggeringBusObj);
if (triggeringBusObj.equals(SourceBusObj)) {
    logInfo("**** triggeringBusObj.equals(SourceBusObj) - TRUE");
} else
    logInfo("**** triggeringBusObj.equals(SourceBusObj) - FALSE");
if (SourceBusObj.equals(triggeringBusObj)) {
    logInfo("**** SourceBusObj.equals(triggeringBusObj) - TRUE");
} else
    logInfo("**** SourceBusObj.equals(triggeringBusObj) - FALSE");

logInfo("*** Setting Source B0 verb to Update");
SourceBusObj.setVerb("Update");
if (triggeringBusObj.equals(SourceBusObj)) {
    logInfo("**** triggeringBusObj.equals(SourceBusObj) - TRUE");
} else
    logInfo("**** triggeringBusObj.equals(SourceBusObj) - FALSE");
if (SourceBusObj.equals(triggeringBusObj)) {
    logInfo("**** SourceBusObj.equals(triggeringBusObj) - TRUE");
} else
    logInfo("**** SourceBusObj.equals(triggeringBusObj) - FALSE");

logInfo("*** Swapping order of Source B0 children");
BusObjArray TempArray = SourceBusObj.getBusObjArray("Attr3");
TempArray.swap(0,1);
if (triggeringBusObj.equals(SourceBusObj)) {
    logInfo("**** triggeringBusObj.equals(SourceBusObj) - TRUE");
} else
    logInfo("**** triggeringBusObj.equals(SourceBusObj) - FALSE");
if (SourceBusObj.equals(triggeringBusObj)) {
    logInfo("**** SourceBusObj.equals(triggeringBusObj) - TRUE");
} else
    logInfo("**** SourceBusObj.equals(triggeringBusObj) - FALSE");

```

The previous code gives the following results:

```

[Msg: Info ***Comparing triggering B0 to empty Source B0]
[Msg: Info **** triggeringBusObj.equals(SourceBusObj) - FALSE]
[Msg: Info **** SourceBusObj.equals(triggeringBusObj) - TRUE]
[Msg: Info *** Copying Source B0 from triggering B0]
[Msg: Info **** triggeringBusObj.equals(SourceBusObj) - TRUE]
[Msg: Info **** SourceBusObj.equals(triggeringBusObj) - TRUE]
[Msg: Info *** Setting Source B0 verb to Update]
[Msg: Info **** triggeringBusObj.equals(SourceBusObj) - TRUE]
[Msg: Info **** SourceBusObj.equals(triggeringBusObj) - TRUE]
[Msg: Info *** Swapping order of Source B0 children]
[Msg: Info **** triggeringBusObj.equals(SourceBusObj) - FALSE]
[Msg: Info **** SourceBusObj.equals(triggeringBusObj) - FALSE]

```

equalsShallow()

Compare this business object's attribute values with those in the input business object, excluding child business objects from the comparison.

Syntax

```
boolean equalsShallow(BusObj inputBusObj)
```

Parameters

inputBusObj A business object to compare with this business object.

Return values

Returns true if the values of all attributes are the same; otherwise, returns false.

Exceptions

CollaborationException—The `equalsShallow()` method can set the following exception type for this exception:

- **ObjectException** – Set if the business object argument is invalid.

See also

`equalKeys()`, `equals()`

Examples

The following example compares attributes of `order2` with attributes of `order1`, excluding the attributes of child business objects, if any.

```
boolean areEqual = order1.equalsShallow(order2);
```

exists()

Check for the existence of a business object attribute with a specified name.

Syntax

```
boolean exists(String attribute)
```

Parameters

attribute The name of an attribute

Return values

Returns true if the attribute exists; returns false if the attribute does not exist.

Examples

The following example checks whether business object `order` has an attribute called `Notes`.

```
boolean notesAreHere = order.exists("Notes");
```

getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString()

Retrieve the value of a single attribute from a business object.

Syntax

```
Object get(String attribute)  
Object get(int position)
```

```

boolean getBoolean(String attribute)
double getDouble(String attribute)
float getFloat(String attribute)
int getInt(String attribute)
long getLong(String attribute)
Object get(String attribute)
BusObj getBusObj(String attribute)
BusObjArray getBusObjArray(String attribute)
String getLongText(String attribute)
String getString(String attribute)

```

Parameters

attribute The name of an attribute.

position is an integer that specifies the ordinal position of an attribute in the business object's attribute list.

Return values

The value of the specified attribute.

Exceptions

CollaborationException—These get methods can set the following exception type for this exception:

- AttributeException—Set if an attribute access problem occurs. For example, this exception can be caused if the collaboration calls `getDouble()` on a String attribute that does not consist of digits or calls `getString()` on a nonexistent attribute.

Notes

These "get" methods retrieve an attribute value from the current business object. They return a copy of the attribute value. They do *not* return an object reference to this attribute in the source business object. Therefore, any change to attribute value in the source business object is *not* made to value that the particular get method returns. Each time one of these get methods is called, it returns a new copy (clone) of the attribute.

These get methods provide the following forms:

- The first form returns a value of the type specified in the method name. For example, `getBoolean()` returns a boolean value, `getBusObj()` returns a BusObj value, `getDouble()` returns a double value, and so on. However, `getLongText()` returns a String object because the WebSphere InterChange Server longtext type is a String object with no maximum size. Use these forms to retrieve attributes with specific basic or WebSphere InterChange Server-defined data types.

These methods provide the ability to access an attribute value by specifying the *name* of the attribute.

- The second form, `get()`, retrieves the value of an attribute of *any* type. You can cast the returned value to the appropriate value of the attribute type.

This method provides the ability to access an attribute value by specifying *either* the *name* of the attribute or the attribute's index *position* within the business object attribute list.

Examples

The following example illustrates how `get()` returns a copy (clone) of the attribute value instead of an object reference:

```
BusObj mySettingBusObj = new BusObj();
BusObj myBusObj = new BusObj();

myBusObj.set("attr1", mySettingBusObj);

BusObj Extract = myBusObj.get("attr1");
```

After this code fragment executes, if you change the Extract business object, mySettingBusObj does *not* change because the get() call returned a copy of the attr1 attribute.

The following example uses getBusObj() to retrieve a child business object containing a customer address from the customer business object and assign it to the variable address.

```
BusObj address = customer.getBusObj("Address");
```

The following example uses getString() to retrieve the value of the CustomerName attribute. The business object variable is sourceCustomer.

```
String customerName = sourceCustomer.getString("CustomerName");
```

The following example uses getInt() to retrieve the Quantity values from two business objects whose variables are item1 and item2. The example then computes the sum of both quantities.

```
int sumQuantity = item1.getInt("Quantity") + item2.getInt("Quantity");
```

The following example retrieves the attribute Item from the business object variable order. The attribute Item is a business object array.

```
BusObjArray items = order.getBusObjArray("Item");
```

The following example gets the CustID attribute value from the source business object and sets the Customer value in the destination business object to match.

```
destination.set("Customer", source.get("CustID"));
```

The following example accesses an attribute value using the attribute's ordinal position within the attribute list:

```
for(i=0; i<maxAttrCount; i++)
{
    String strValue = (String)myBusObj.get(i);
    ...
}
```

getLocale()

Retrieve the locale associated with the business object's data.

Syntax

```
java.util.Locale getLocale()
```

Parameters

None.

Return values

A Java Locale object that contains information about the business object's locale. This Locale object must be an instance of the java.util.Locale class.

Notes

The `getLocale()` method returns the locale associated with the data in a business object. This locale is often different from the collaboration locale in which the collaboration is executing.

See also

`getLocale()` (BaseCollaboration class), `setLocale()`

getType()

Retrieve the name of the business object definition on which this business object was based.

Syntax

```
String getType()
```

Return values

The name of a business object definition.

Notes

The type of a business object, in terms of this method, is the name of the business object definition from which the business object was created.

Examples

The following example retrieves the type of a business object called `sourceShipTo`.

```
String typeName = sourceShipTo.getType();
```

The following example copies a triggering event into a new business object of the appropriate type.

```
BusObj source = new BusObj(triggeringBusObj.getType());
```

getVerb()

Retrieve this business object's verb.

Syntax

```
String getVerb()
```

Return values

The name of a verb, such as Create, Retrieve, Update, or Delete.

Notes

This method is useful for scenarios that handle multiple types of incoming events. The first action node in a scenario calls `getVerb()`. The outgoing transition links from that action node then test the contents of the returned string, so that each outgoing transition link is the start of an execution path that handles one of the possible verbs.

Examples

The following example obtains the verb from a business object called `orderEvent` and assigns it to a variable called `orderVerb`.

```
String orderVerb = orderEvent.getVerb();
```

isBlank()

Find out whether the value of an attribute is set to a zero-length string.

Syntax

```
boolean isBlank(String attribute)
```

Parameters

attribute The name of an attribute.

Return values

Returns true if the attribute value is a zero-length string; otherwise, returns false.

Notes

A zero-length string can be compared to the string `""`. It is different from a null, whose presence is detected by the `isNull()` method.

If a collaboration needs to retrieve an attribute value and then do something with it, it can call `isBlank()` and `isNull()` to check that it has a value before retrieving the value.

Examples

The following example checks whether the `Material` attribute of the `sourcePaperClip` business object is a zero-length string.

```
boolean key = sourcePaperClip.isBlank("Material");
```

isKey()

Find out whether a business object's attribute is defined as a key attribute.

Syntax

```
boolean isKey(String attribute)
```

Parameters

attribute The name of an attribute.

Return values

Returns true if the attribute is a key attribute; returns false if it is not a key attribute.

Examples

The following example determines whether the `CustID` attribute of the `customer` business object is a key attribute.

```
boolean keyId = (customer.isKey("CustID"));
```

isNull()

Find out whether the value of a business object's attribute is null.

Syntax

```
boolean isNull(String attribute)
```

Parameters

attribute The name of an attribute.

Return values

Returns true if the attribute value is null; returns false if it is not null.

Notes

A null indicates no value, in contrast to a zero-length string value, which is detected by calling `isBlank()`. Test an object with `isNull()` before using it, because if the object is null, the operation could fail.

An attribute value can be null under these circumstances:

- The attribute value was explicitly set to null.

An attribute value can be set to null using the `set()` method.

- The attribute value was never set.

When a collaboration uses the `new()` method to create a new business object, all attribute values are initialized to null. If the attribute value has not been set between the time of creation and the time of the `isNull()` call, the value is still null.

- The null was inserted during mapping.

When a collaboration processes a business object received from a connector, the mapping process might insert a null value. The mapping process converts the application-specific business object received from the connector to the generic business object handled by the collaboration. For each attribute in the generic business object that has no equivalent in the application-specific object, the map inserts a null value.

Tip: Always call `isNull()` before performing an operation on an attribute that is a child business object or child business object array, because Java does not allow operations on null objects.

Examples

The following example checks whether the `Material` attribute of the `sourcePaperClip` business object has a null value.

```
boolean key = sourcePaperClip.isNull("Material");
```

The following example checks whether the `CustAddr` attribute of the `contract1` business object is null before retrieving it. The attribute retrieval proceeds only if the `isNull()` check is false, showing that the attribute is not null.

```
if (! contract1.isNull("CustAddr"))
{
    BusObj customerAddress = contract1.getBusObj("CustAddr");
    // do something with the "customerAddress" business object
}
```

isRequired()

Find out whether a business object's attribute is defined as a required attribute.

Syntax

```
boolean isRequired(String attribute)
```

Parameters

attribute The name of an attribute.

Return values

Returns true if the attribute is required; returns false if it is not required.

Notes

If an attribute is defined as required, it must have a value and the value must not be a null.

Examples

The following example logs a warning if a required attribute has a null value.

```
if ( (customer.isRequired("Address"))
    && (customerBusObj.isNull("Address")) )
    {
        logWarning(12, "Address is required and cannot be null.");
    }
else
    {
        // do something else
    }
```

keysToString()

Retrieve the values of a business object's primary key attributes as a string.

Syntax

```
String keysToString()
```

Return values

A String object containing all the key values in a business object, concatenated, and ordered by the ordinal value of the attributes.

Notes

The output from this method contains the name of the attribute and its value. Multiple values are primary key attribute values, concatenated and separated by spaces. For example, if there is one primary key attribute, SS#, the output might be:

```
SS#=100408394
```

If the primary key attributes are FirstName and LastName, the output might be:

```
FirstName=Nina LastName=Silk
```

Examples

The following example returns the values of key attributes of the business object represented by the variable name `fromOrder`.

```
String keyValues = fromOrder.keysToString();
```

set()

Set a business object's attribute to a specified value of a particular data type.

Syntax

```
void set(String attribute, Object value)  
void set(int position, Object value)  
void set(String attribute, boolean value)  
void set(String attribute, double value)  
void set(String attribute, float value)  
void set(String attribute, int value)  
void set(String attribute, long value)  
void set(String attribute, Object value)  
void set(String attribute, String value)
```

Parameters

<i>attribute</i>	The name of the attribute to set.
<i>position</i>	An integer that specifies the ordinal position of an attribute in the business object's attribute list.
<i>value</i>	An attribute value.

Exceptions

`CollaborationException`—The `set()` method can set the following exception type for this exception:

- `AttributeException`—Set if an attribute access problem occurs.

Notes

These `set()` methods set an attribute value in the current business object. These methods set an object reference when it assigns the value to the attribute. It does *not* clone the attribute value from the source business object. Therefore, any changes to *value* in the source business object are also made to the attribute in the business object that calls `set()`.

The `set()` method provides the following forms:

- The first form sets a value of the type specified by the method's second parameter type. For example, `set(String attribute, boolean value)` sets an attribute with a boolean value, `set(String attribute, double value)` sets an attribute with a double value, and so on. Use this form to set attributes with specific basic or WebSphere InterChange Server-defined data types.

These methods provide the ability to access an attribute value by specifying the *name* of the attribute.

- The second form sets the value of an attribute of *any* type. You can send in any data type as the attribute value because the attribute-value parameter is of type `Object`. For example, to set an attribute that is of `BusObj` or `LongText` object, use this form of the method and pass in the `BusObj` or `LongText` object as the attribute value.

This form of the `set()` method provides the ability to access an attribute value by specifying *either* the *name* of the attribute or the attribute's index *position* within the business object attribute list.

Examples

The following example sets the `LName` attribute in `toCustomer` to the value `Smith`.

```
toCustomer.set("LName", "Smith");
```

The following example illustrates how `set()` assigns an object reference instead of cloning the value:

```
BusObj BusObj myBusObj = new BusObj();  
BusObj mySettingBusObj = new BusObj();  
myBusObj.set("attr1", mySettingBusObj);
```

After this code fragment executes, the `attr1` attribute of `myBusObj` is set to the `mySettingBusObj` business object. If `mySettingBusObj` is changed in any way, `myBusObj.attr1` is changed in the exact manner because `set()` makes an object reference to `mySettingBusObj` when it sets the `attr1` attribute; it does *not* create a static copy of `mySettingBusObj`.

The following example sets an attribute value using the attribute's ordinal position within the attribute list:

```
for(i=0; i<maxAttrCount; i++)  
{  
    myBusObj.set(i, strValue);  
    ...  
}
```

setDefaultAttrValues()

Set all attributes to their default values.

Syntax

```
void setDefaultAttrValues()
```

Notes

A business object definition can include default values for attributes. The method sets the values of this business object's attributes to the values specified as defaults in the definition.

Examples

The following example sets the values of the `PaperClip` business object to their default values:

```
PaperClip.setDefaultAttrValues();
```

setKeys()

Set the values of this business object's key attributes to the values of the key attributes in another business object.

Syntax

```
void setKeys(BusObj inputBusObj)
```

Parameters

inputBusObj The business object whose values are used to set values of another business object

Exceptions

CollaborationException—The `setKeys()` method can set one of the following exception types for this exception:

- `AttributeException` – Set if an attribute access problem occurs.
- `ObjectException` – Set if the business object argument is invalid.

Examples

The following example sets the key values in the business object `helpdeskCustomer` to the key values in the business object `ERPCustomer`.

```
helpdeskCustomer.setKeys(ERPCustomer);
```

setLocale()

Set the locale of the current business object.

Syntax

```
void setLocale(java.util.Locale locale)
```

Parameters

locale The Java Locale object that contains the information about the locale to assign to the business object. This Locale object must be an instance of the `java.util.Locale` class.

Return values

None.

Notes

The `setLocale()` method assigns a locale to the data associated with a business object. This locale might be different from the collaboration locale in which the collaboration executes.

See also

`getLocale()`

setVerb()

Set the verb of a business object.

Syntax

```
void setVerb(String verb)
```

Parameters

verb The verb of the business object.

Notes

This method is used in mapping business objects.

Do not use this method in a collaboration template, where you must set the verb of an outgoing business object interactively by filling in the properties of a service call.

Examples

The following example sets the verb Delete on the business object `contactAddress`.

```
contactAddress.setVerb("Delete");
```

setWithCreate()

Set a business object's attribute to a value of a specified data type.

Syntax

```
void setWithCreate(String attributeName, BusObj busObj)
void setWithCreate(String attributeName, BusObjArray busObjArray)
void setWithCreate(String attributeName, Object value)
```

Parameters

<i>attributeName</i>	The name of the attribute to set.
<i>busObj</i>	The business object to insert into the target attribute.
<i>busObjArray</i>	The business object array to insert into the target attribute.
<i>value</i>	The object to insert into the target attribute. This object needs to be one of the following types: <code>BusObj</code> , <code>BusObjArray</code> , <code>Object</code> .

Exceptions

`CollaborationException`—The `setWithCreate()` method can set the following exception type for this exception:

- `AttributeException`—Set if an attribute access problem occurs.

Notes

If the object provided is a `BusObj` and the target attribute contains multi-cardinality child business object, the `BusObj` is appended to the `BusObjArray` as its last element. If the target attribute contains a `BusObj`, however, this business object replaces the previous value.

Examples

The following example sets an attribute called `ChildAttrAttr` to the value 5. The attribute is found in a business object contained in `myBO`'s attribute, `ChildAttr`. If the `childAttr` business object does not exist at the time of the call, this method call creates it.

```
myBO.setWithCreate("childAttr.childAttrAttr", "5");
```

toString()

Return the values of all attributes in a business object as a string.

Syntax

```
String toString()
```

Return values

A String object containing all attribute values contained in a business object.

Notes

The string that results from a call to this method is similar to the following example:

```
Name: GenEmployee  
Verb: Create  
Type: AfterImage  
Attributes: (Name, Type, Value)
```

```
LastName:String, Davis  
FirstName:String, Miles  
SS#:String, 041-33-8989  
Salary:Float, 15.00  
ObjectEventId:String, MyConnector_922323619411_1
```

Examples

The following example returns a string containing the attribute values of the business object variable `fromOrder`.

```
String values = fromOrder.toString();
```

validData()

Checks whether a specified value is a valid data type for a specified attribute.

Syntax

```
boolean validData(String attributeName, Object value)  
boolean validData(String attributeName, BusObj value)  
boolean validData(String attributeName, BusObjArray value)  
boolean validData(String attributeName, String value)  
boolean validData(String attributeName, long value)  
boolean validData(String attributeName, int value)  
boolean validData(String attributeName, double value)  
boolean validData(String attributeName, float value)  
boolean validData(String attributeName, boolean value)
```

Parameters

attributeName The attribute.

value The value.

Returns

true or false (boolean return)

Notes

Checks the compatibility of the value passed in with the target attribute (as specified by *attributeName*). These are the criteria:

for primitive types (String, long, int, double, float, boolean)	the value must be convertible to the data type of the attribute
---	---

for a BusObj	the value must have the same type as that of the target attribute
for a BusObjArray	the value must point to a BusObj or BusObjArray with the same (business object definition) type as that of the attribute
for an Object	the value must be of type String, BusObj, or BusObjArray. The corresponding validation rules are then applied.

Deprecated method

Some methods in the BusObj class were supported in earlier versions but are no longer supported. Use of these **deprecated methods** does not generate an error. However, it is highly recommended that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 83 lists the deprecated methods for the BusObj class and the replacement methods. If you have not used Process Designer before, ignore this section.

Table 83. Deprecated methods, BusObj class

Former method	Replacement
getCount()	BusObjArray.size()
getKeys()	keysToString()
getValues()	toString()
not	standard Java NOT operator, "!"
set(BusObj <i>inputBusObj</i>)	copy()
All methods that took a child business object or child business object array as an input argument	Get a handle to the child business object or business object array and use the methods of the BusObj or BusObjArray class

The setVerb() method, which was previously listed as deprecated, is now restored for use in mapping. Do not use it within a collaboration.

Chapter 24. BusObjArray class

The methods documented in this chapter operate on objects of the BusObjArray class. They are defined on the WebSphere InterChange Server-defined class BusObjArray. The BusObjArray class encapsulates an array of business objects. In a hierarchical business object, an attribute is a reference to an array of child business objects when its cardinality is equal to n. Operations on the BusObjArray class can return either a BusObjArray object or an actual array of business objects.

Table 84 lists the methods documented in this chapter.

Table 84. BusObjArray method summary

Method	Description	Page
addElement()	Add a business object to this business object array.	386
duplicate()	Create a business object array (a BusObjArray object) exactly like this one.	386
elementAt()	Retrieve a single business object by specifying its position in this business object array.	387
equals()	Compare another business object array with this one.	387
getElements()	Retrieve the contents of this business object array.	387
getLastIndex()	Retrieve the last available index from a business object array.	388
max()	Retrieve the maximum value for the specified attribute among all elements in this business object array.	388
maxBusObjArray()	Returns the business objects that have the maximum value for the specified attribute, as a business object array (BusObjArray object).	389
maxBusObjs()	Returns the business objects that have the maximum value for the specified attribute, as an array of BusObj objects.	390
min()	Retrieve the minimum value for the specified attribute among the business objects in this array.	391
minBusObjArray()	Returns the business objects that have the minimum value for the specified attribute, as a BusObjArray object.	392
minBusObjs()	Returns the business objects that have the minimum value for the specified attribute, as an array of BusObj objects.	393
removeAllElements()	Remove all elements from this business object array.	394
removeElement()	Delete an element from a business object array.	394
removeElementAt()	Remove an element at a particular position in this business object array.	394
setElementAt()	Set the value of a business object in a business object array.	395

Table 84. BusObjArray method summary (continued)

Method	Description	Page
size()	Return the number of elements in this business object array.	395
sum()	Adds the values of the specified attribute for all business objects in this business object array.	396
swap()	Reverse the positions of two business objects in a business object array. Keep in mind that the first element in the array is zero (0), the second is 1, the third is 2, and so on.	396
toString()	Retrieve the values in this business object array and return them in a single string.	397

addElement()

Add a business object to this business object array.

Syntax

```
void addElement(BusObj element)
```

Parameters

element A business object to add to the array.

Exceptions

CollaborationException—The addElement() method can set the following exception type for this exception:

- AttributeException – Set if the element is not valid.

Examples

The following example uses the getBusObjArray() method to retrieve an array of business objects called itemList from the business object order. The array is assigned to items, and then a new business object is added to items.

```
BusObjArray items = order.getBusObjArray("itemList");  
items.addElement(new BusObj("oneItem"));
```

duplicate()

Create a business object array (a BusObjArray object) exactly like this one.

Syntax

```
BusObjArray duplicate()
```

Return values

A business object array.

Examples

The following example duplicates the items array, creating newItem.

```
BusObjArray newItem = items.duplicate();
```

elementAt()

Retrieve a single business object by specifying its position in this business object array.

Syntax

```
BusObj elementAt(int index)
```

Parameters

index The array element to retrieve. The first element in the array is zero (0), the second is 1, the third is 2, and so on.

Exceptions

CollaborationException—The `elementAt()` method can set the following exception type for this exception:

- `AttributeException` – Set if the element is not valid.

Examples

The following example retrieves the 11th business object in the `items` array and assigns it to the `Item` variable.

```
BusObj Item = items.elementAt(10);
```

equals()

Compare another business object array with this one.

Syntax

```
boolean equals(BusObjArray inputBusObjArray)
```

Parameters

inputBusObjArray

A business object array to compare with this business object array.

Notes

The comparison between the two business object arrays checks the number of elements and their attribute values.

Examples

The following example uses `equals()` to set up a conditional loop, the inside of which is not shown.

```
if (items.equals(newItems))  
{  
  ...  
}
```

getElements()

Retrieve the contents of this business object array.

Syntax

```
BusObj[] getElements()
```

Exceptions

CollaborationException—The `getElements()` method can set the following exception type for this exception:

- `ObjectException`—Set if one of the elements is not valid.

Examples

The following example prints the elements of the items array.

```
BusObj[] elements = items.getElements();
for (i=0, i<size; i++)
{
    trace(1, elements[i].toString());
}
```

getLastIndex()

Retrieve the last available index from a business object array.

Syntax

```
int getLastIndex()
```

Returns

The last index to the last element in this `BusObjArray`.

Notes

Note that, previously, the `size()` method was used to do this. That is, the user would use the `size()` of the business object array to retrieve the last index available in a `BusObjArray`. Unfortunately, this approach yields incorrect data if the `BusObjArray` contains gaps.

Like all Java arrays, `BusObjArray` is a zero relative array. This means that the `size()` method will return 1 greater than the `getLastIndex()` method.

Examples

The following example retrieves the last index in the business object array.

```
int lastElementIndex = items.getLastIndex();
```

max()

Retrieve the maximum value for the specified attribute among all elements in this business object array.

Syntax

```
String max(String attr)
```

Parameters

attr A variable that refers to an attribute in the business object. The attribute must be one of these types: `String`, `LongText`, `int`, `float`, and `double`.

Returns

The maximum value of the specified attribute in the form of a string, or null if the value for that attribute is null for all elements in this `BusObjArray`.

Exceptions

`UnknownAttributeException`—When the specified attribute is not a valid attribute in the business objects passed in.

`UnsupportedAttributeTypeException`—When the type of the specified attribute is not one of the supported attribute types listed in the note section.

All of the above exceptions are subclassed from `CollaborationException`. The `max()` method can set the following exception type for these exceptions: `AttributeException`.

Notes

The `max()` method looks for the maximum value for the specified attribute among the business objects in this `BusObjArray`. For example, if three employee objects are used, and the attribute is “Salary” which is of type “Float,” it will return the string representing the largest salary.

If the value of the specified attribute for an element in `BusObjArray` is null, then that element is ignored. If the value of the specified attribute is null for all elements, then null is returned.

When the attribute type is of type `String`, `max()` returns the attribute value that is the longest string lexically.

Examples

```
String maxSalary = items.max("Salary");
```

`maxBusObjArray()`

Returns the business objects that have the maximum value for the specified attribute, as a business object array (`BusObjArray` object).

Syntax

```
BusObjArray maxBusObjArray(String attr)
```

Parameters

attr A `String`, `LongText`, `int`, `float`, or `double` variable that refers to an attribute in a business object in the business object array.

Returns

A list of business objects in the form of `BusObjArray` or null.

Exceptions

`UnknownAttributeException` – When the specified attribute is not a valid attribute in the business objects passed in.

`UnsupportedAttributeTypeException` – When the type of the specified attribute is not one of the supported attribute types listed in the note section.

All of the above exceptions are subclassed from `CollaborationException`. The `maxBusObjArray()` method can set the following exception type for these exceptions: `AttributeException`.

Notes

The `maxBusObjArray()` method finds one or more business objects with the maximum value for the specified attribute, and returns these business objects in a `BusObjArray` object.

For example, suppose that this is a business object array containing `Employee` business objects and that the input argument is the attribute `Salary`, a `Float`. The method determines the largest value for `Salary` in all the `Employee` business objects and returns the business object that contains that value. If multiple business objects have that largest `Salary` value, the method returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type `String`, the method returns the longest string lexically.

Examples

```
BusObjArray boarrayWithMaxSalary = items.maxBusObjArray("Salary");
```

maxBusObjs()

Returns the business objects that have the maximum value for the specified attribute, as an array of `BusObj` objects.

Syntax

```
BusObj[] maxBusObjs(String attr)
```

Parameters

attr A `String`, `LongText`, `int`, `float`, or `double` variable that refers to an attribute in the business object.

Returns

A list of business objects in the form of a `BusObj[]` or `null`.

Exceptions

`UnknownAttributeException` – When the specified attribute is not a valid attribute in the business objects passed in.

`UnsupportedAttributeTypeException` – When the type of the specified attribute is not one of the supported attribute types listed in the note section.

All of the above exceptions are subclassed from `CollaborationException`. The `maxBusObjs()` method can set the following exception type for these exceptions: `AttributeException`.

Notes

The `maxBusObjs()` method finds one or more business objects with the maximum value for the specified attribute, and returns these business objects as an array of `BusObj` objects.

For example, suppose that this is a business object array containing `Employee` business objects and that the input argument is the attribute `Salary`, a `Float`. The method determines the largest value for `Salary` in all the `Employee` business objects and returns the business object that contains that value. If multiple business objects have that largest `Salary` value, the method returns all of those business objects.

A business object is ignored if the specified attribute contains `null`. If the value is `null` in all business objects in the array, `null` is returned.

When the attribute is of type `String`, the method returns the longest string lexically.

Examples

```
BusObj[] bosWithMaxSalary = items.maxBusObjs("Salary");
```

min()

Retrieve the minimum value for the specified attribute among the business objects in this array.

Syntax

```
String min(String attr)
```

Parameters

attr A `String`, `LongText`, `int`, `float`, or `double` variable that refers to an attribute in the business object.

Returns

The minimum value of the specified attribute in the form of a string, or `null` if the value for that attribute is `null` for all elements in this `BusObjArray`.

Exceptions

`UnknownAttributeException` – When the specified attribute is not a valid attribute in the business objects passed in.

`UnsupportedAttributeTypeException` – When the type of the specified attribute is not one of the supported attribute types listed in the note section.

All of the above exceptions are subclassed from `CollaborationException`. The `min()` method can set the following exception type for these exceptions: `AttributeException`.

Notes

The `min()` method looks for the minimum value for the specified attribute among the business objects in this business object array.

For example, suppose that this is a business object array containing `Employee` business objects and that the input argument is the attribute `Salary`, a `Float`. The

method determines the smallest value for Salary in all the Employee business objects and returns the business object that contains that value. If multiple business objects have that lowest Salary value, the method returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type String, the method returns the shortest string lexically.

Examples

```
String minSalary = items.min("Salary");
```

minBusObjArray()

Returns the business objects that have the minimum value for the specified attribute, as a BusObjArray object.

Syntax

```
BusObjArray minBusObjArray(String attr)
```

Parameters

attr A String, LongText, int, float, or double variable that refers to an attribute in the business object.

Returns

A list of business objects in the form of BusObjArray or null.

Exceptions

`UnknownAttributeException` – When the specified attribute is not a valid attribute in the business objects passed in.

`UnsupportedAttributeTypeException` – When the type of the specified attribute is not one of the supported attribute types listed in the note section.

All of the above exceptions are subclassed from `CollaborationException`. The `minBusObjArray()` method can set the following exception type for these exceptions: `AttributeException`.

Notes

The `minBusObjArray()` method finds one or more business objects with the minimum value for the specified attribute, and returns these business objects in a `BusObjArray` object.

For example, suppose that this is a business object array containing Employee business objects and that the input argument is the attribute Salary, a Float. The method determines the smallest value for Salary in all the Employee business objects and returns the business object that contains that value. If multiple business objects have that smallest Salary value, the method returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type String, the method returns the shortest string lexically.

Examples

```
BusObjArray boarrayWithMinSalary = items.minBusObjArray("Salary");
```

minBusObjs()

Returns the business objects that have the minimum value for the specified attribute, as an array of BusObj objects.

Syntax

```
BusObj[] minBusObjs(String attr)
```

Parameters

attr A String, LongText, int, float, or double variable that refers to an attribute in the business object.

Returns

A list of business objects in the form of a BusObj[] or null.

Exceptions

`UnknownAttributeException` – When the specified attribute is not a valid attribute in the business objects passed in.

`UnsupportedAttributeTypeException` – When the type of the specified attribute is not one of the supported attribute types listed in the note section.

All of the above exceptions are subclassed from `CollaborationException`. The `minBusObjs()` method can set the following exception type for these exceptions: `AttributeException`.

Notes

The `minBusObjs()` method finds one or more business objects with the maximum value for the specified attribute, and returns these business objects as an array of BusObj objects.

For example, suppose that this is a business object array containing Employee business objects and that the input argument is the attribute Salary, a Float. The method determines the smallest value for Salary in all the Employee business objects and returns the business object that contains that value. If multiple business objects have that smallest Salary value, the method returns all of those business objects.

A business object is ignored if the specified attribute contains null. If the value is null in all business objects in the array, null is returned.

When the attribute is of type String, the method returns the shortest string lexically.

Examples

```
BusObj[] bosWithMinSalary = items.minBusObjs("Salary");
```

removeAllElements()

Remove all elements from this business object array.

Syntax

```
void removeAllElements()
```

Examples

The following example removes all elements of the array `items`.

```
items.removeAllElements();
```

removeElement()

Delete an element from a business object array.

Syntax

```
void removeElement(BusObj element)
```

Parameters

elementReference A variable that refers to an element of the array.

Exceptions

`CollaborationException`—The `removeElement()` method can set the following exception type for this exception:

- `AttributeException` – Set if the element is not valid.

Notes

After you delete an element from the array, the array resizes, changing the indexes of existing elements.

Examples

The following example deletes the element `Child1` from the business object array `items`.

```
items.removeElement(Child1);
```

removeElementAt()

Remove an element at a particular position in this business object array.

Syntax

```
void removeElementAt(int index)
```

Notes

After an element is removed from the array, the array resizes, possibly changing the indexes of existing elements.

Parameters

index An integer representing the element's position within the array. The first element in an array is at position zero (0), the second element is at position 1, the third is at position 2, and so forth.

Exceptions

`CollaborationException`—The `removeElementAt()` method can set the following exception type for this exception:

- `AttributeException` – Set if the element is not valid.

Examples

The following example deletes the sixth business object in the array `items`.

```
items.removeElementAt(5);
```

setElementAt()

Set the value of a business object in a business object array.

Syntax

```
void setElementAt (int index, BusObj element)
```

Parameters

index An integer representing the array position. The first element in the array is zero (0), the second is 1, the third is 2, and so on.

inputBusObj The business object containing the values to which you want to set the array element.

Exceptions

`CollaborationException`—The `setElementAt()` method can set the following exception type for this exception:

- `AttributeException` – Set if the element is not valid.

Notes

This method sets the values of the business object at a specified array position to the values of an input business object.

Examples

The following example creates a new business object of type `Item` and adds it to the array `items`, as the fourth element.

```
items.setElementAt(3, new BusObj("Item"));
```

size()

Return the number of elements in this business object array.

Syntax

```
int size()
```

Examples

The following example returns the number of elements in the array `items`.

```
int size = items.size();
```

sum()

Adds the values of the specified attribute for all business objects in this business object array.

Syntax

```
double sum(String attrName)
```

Parameters

attr A variable that refers to an attribute in the business object. The attribute must be of type `int`, `float`, or `double`.

Returns

The sum of the specified attribute from the list of the business objects.

Exceptions

`UnknownAttributeException` – When the specified attribute is not a valid attribute in the business objects passed in.

`UnsupportedAttributeTypeException` – When the type of the specified attribute is not one of the supported attribute types listed in the note section.

All of the above exceptions are subclassed from `CollaborationException`. The `sum()` method can set the following exception type for these exceptions: `AttributeException`.

Examples

```
double sumSalary = items.sum("Salary");
```

swap()

Reverse the positions of two business objects in a business object array. Keep in mind that the first element in the array is zero (0), the second is 1, the third is 2, and so on.

Syntax

```
void swap(int index1, int index2)
```

Parameters

index1 The array position of one element you want to swap.

index2 The array position of the other element you want to swap.

Examples

The following example uses `swap()` to reverse the positions of `BusObjA` and `BusObjC` in the following array:

BusObjA	BusObjB	BusObjC
---------	---------	---------

```
swap(0,2);
```

The result of the call to `swap()` is the following array:

BusObjC	BusObjB	BusObjA
---------	---------	---------

toString()

Retrieve the values in this business object array and return them in a single string.

Syntax

```
String toString()
```

Examples

The following example uses `toString()` to retrieve the contents of the `items` business object array and then uses `logInfo()` to write the contents to the log file.

```
logInfo(items.toString());
```

Chapter 25. CwDBConnection class

The CwDBConnection class provides methods for executing SQL queries in a database. Queries are performed through a connection, which is obtained from a connection pool. To instantiate this class, you must call `getDBConnection()` in the `BaseCollaboration` class. All collaborations are derived or subclassed from `BaseCollaboration` so they have access to `getDBConnection()`.

Table 85 summarizes the methods in the CwDBConnection class.

Table 85. CwDBConnection method summary

Method	Description	Page
<code>beginTransaction()</code>	Begins an explicit transaction for the current connection.	399
<code>commit()</code>	Commits the active transaction associated with the current connection.	400
<code>executeSQL()</code>	Executes a static SQL query by specifying its syntax and an optional parameter array.	402
<code>executePreparedSQL()</code>	Executes a prepared SQL query by specifying its syntax and an optional parameter array.	401
<code>executeStoredProcedure()</code>	Executes an SQL stored procedure by specifying its name and parameter array.	404
<code>getUpdateCount()</code>	Returns the number of rows affected by the last write operation to the database.	405
<code>hasMoreRows()</code>	Determines whether the query result has more rows to process.	406
<code>inTransaction()</code>	Determines whether a transaction is in progress in the current connection.	406
<code>isActive()</code>	Determines whether the current connection is active.	407
<code>nextRow()</code>	Retrieves the next row from the query result.	407
<code>release()</code>	Releases use of the current connection, returning it to its connection pool.	408
<code>rollback()</code>	Rolls back the active transaction associated with the current connection.	409

beginTransaction()

Begins an explicit transaction for the current connection.

Syntax

```
void beginTransaction()
```

Parameters

None.

Return values

None.

Exceptions

CwDBConnectionException—If a database error occurs.

Notes

The `beginTransaction()` method marks the beginning of a new explicit transaction in the current connection. The `beginTransaction()`, `commit()` and `rollback()` methods together provide management of transaction boundaries for an explicit transaction. This transaction contains SQL queries, which include the SQL statements `INSERT`, `DELETE`, or `UPDATE`, and a stored procedure that includes one of these SQL statements.

Important: Only use `beginTransaction()` if the connection uses explicit transaction bracketing. If the connection uses implicit transaction bracketing, use of `beginTransaction()` results in a `CwDBTransactionException` exception.

Before beginning an explicit transaction, you must create a `CwDBConnection` object with the `getDBConnection()` method from the `BaseCollaboration` class. Make sure that this connection uses explicit transaction bracketing

Examples

The following example uses a transaction to execute a query for inserting rows into a table in the database associated with connections in the `CustDBConnPool`.

```
CwDBConnection connection = getDBConnection("CustDBConnPool", false);

// Begin a transaction
connection.beginTransaction();

// Insert a row
connection.executeSQL("insert...");

// Commit the transaction
connection.commit();

// Release the connection
connection.release();
```

See also

“Managing the transaction” on page 224

`commit()`, `getDBConnection()`, `beginTransaction()`, `rollback()`

commit()

Commits the active transaction associated with the current connection.

Syntax

```
void commit()
```

Parameters

None.

Return values

None.

Exceptions

`CwDBConnectionException` – If a database error occurs.

Notes

The `commit()` method ends the active transaction by committing any changes made to the database associated with the current connection. The `beginTransaction()`, `commit()` and `rollback()` methods together provide management of transaction boundaries for an explicit transaction. This transaction contains SQL queries, which include the SQL statements `INSERT`, `DELETE`, or `UPDATE`, and a stored procedure that includes one of these SQL statements.

Important: Only use `commit()` if the connection uses explicit transaction bracketing. If the connection uses implicit transaction bracketing, use of `commit()` results in a `CwDBTransactionException` exception. If you do not end an explicit transaction with `commit()` (or `rollback()`) before the connection is released, InterChange Server implicitly ends the transaction based on the success of the collaboration. If the collaboration is successful, ICS commits this database transaction. If the collaboration is *not* successful, ICS implicitly rolls back the database transaction. Regardless of the success of the collaboration, ICS logs a warning.

Before beginning an explicit transaction, you must create a `CwDBConnection` object with the `getDBConnection()` method from the `BaseCollaboration` class. Make sure that this connection uses explicit transaction bracketing

Examples

For an example of committing a transaction, see the example for `"beginTransaction()"` on page 399.

See also

"Managing the transaction" on page 224

`"beginTransaction()"` on page 399, `getDBConnection()`, `inTransaction()`, `rollback()`

`executePreparedSQL()`

Executes a prepared SQL query by specifying its syntax and an optional parameter array.

Syntax

```
void executePreparedSQL(String query)
void executePreparedSQL(String query, Vector queryParameters)
```

Parameters

query A string representation of the SQL query to execute in the database.

queryParameters A Vector object of arguments to pass to parameters in the SQL query.

Return values

None.

Exceptions

`CwDBSQLException` – If a database error occurs.

Notes

The `executePreparedSQL()` method sends the specified *query* string as a prepared SQL statement to the database associated with the current connection. The first time it executes, this query is sent as a string to the database, which compiles the string into an executable form (called a prepared statement), executes the SQL statement, and returns this prepared statement to `executePreparedSQL()`. The `executePreparedSQL()` method saves this prepared statement in memory. Use `executePreparedSQL()` for SQL statements that you need to execute multiple times. The `executeSQL()` method does *not* save the prepared statement and is therefore useful for queries you need to execute only once.

Important: Before executing a query with `executePreparedSQL()`, you must obtain a connection to the desired database by generating a `CwDBConnection` object with the `getDBConnection()` method from the `BaseDLM` class.

The SQL statements you can execute include the following (as long as you have the necessary database permissions):

- The `SELECT` statement to request data from one or more database tables
Use the `hasMoreRows()` and `nextRow()` methods to access the retrieved data.
- SQL statements that modify data in the database
 - `INSERT`
 - `DELETE`
 - `UPDATE`

If the connection uses explicit transaction bracketing, you must explicitly start each transaction with `beginTransaction()` and end it with either `commit()` or `rollback()`.

- The `CALL` statement to execute a prepared stored procedures with the limitation that this stored procedure *cannot* use any `OUT` parameters
To execute stored procedures with `OUT` parameters, use the `executeStoredProcedure()` method. For more information, see “Calling stored procedures with `executeStoredProcedure()`” on page 221.

See also

“Executing prepared queries” on page 217

`beginTransaction()` on page 399, `commit()` on page 400, `executeSQL()`, `executeStoredProcedure()`, `getDBConnection()`, `hasMoreRows()`, `nextRow()`, `rollback()`

executeSQL()

Executes a static SQL query by specifying its syntax and an optional parameter array.

Syntax

```
void executeSQL(String query)
void executeSQL(String query, Vector queryParameters)
```

Parameters

- query* A string representation of the SQL query to execute in the database.
- queryParameters* A Vector object of arguments to pass to parameters in the SQL query.

Return values

None.

Exceptions

CwDBSQLException – If a database error occurs.

Notes

The `executeSQL()` method sends the specified *query* string as a static SQL statement to the database associated with the current connection. This query is sent as a string to the database, which compiles the string into an executable form and executes the SQL statement, without saving this executable form. Use `executeSQL()` for SQL statements that you need to execute only once. The `executePreparedSQL()` method saves the executable form (called a prepared statement) and is therefore useful for queries you need to execute multiple times.

Important: Before executing a query with `executeSQL()`, you must obtain a connection to the desired database by generating a `CwDBConnection` object with the `getDBConnection()` method from the `BaseDLM` class.

The SQL statements you can execute include the following (as long as you have the necessary database permissions):

- The `SELECT` statement to request data from one or more database tables
Use the `hasMoreRows()` and `nextRow()` methods to access the retrieved data.
- SQL statements that modify data in the database
 - `INSERT`
 - `DELETE`
 - `UPDATE`

If the connection uses explicit transaction bracketing, you must explicitly start each transaction with `beginTransaction()` and end it with either `commit()` or `rollback()`.

- The `CALL` statement to statically execute a stored procedure with the limitation that this stored procedure *cannot* use any `OUT` parameters
To execute stored procedures with `OUT` parameters, use the `executeStoredProcedure()` method. For more information, see “Calling stored procedures with `executeStoredProcedure()`” on page 221.

Examples

The following example executes a query for inserting rows into an accounting database whose connections reside in the `AccntConnPool` connection pool.

```
CwDBConnection connection = getDBConnection("AccntConnPool");  
  
// Begin a transaction  
connection.beginTransaction();
```

```
// Insert a row
connection.executeSQL("insert...");

// Commit the transaction
connection.commit();

// Release the database connection
connection.release();
```

For a more complete code sample that selects data from a database table, see “Executing static queries that return data (SELECT)” on page 214.

See also

“Executing static queries” on page 213

`executePreparedSQL()` on page 401, `executeStoredProcedure()`, `getDBConnection()`, `hasMoreRows()`, `nextRow()`

executeStoredProcedure()

Executes an SQL stored procedure by specifying its name and parameter array.

Syntax

```
void executeStoredProcedure(String storedProcedure,
                           Vector storedProcParameters)
```

Parameters

storedProcedure

The name of the SQL stored procedure to execute in the database.

storedProcParameters

A Vector object of parameters to pass to the stored procedure. Each parameter is an instance of the `CwDBStoredProcedureParam` class. For more information on how to pass parameters through this array, see “Calling stored procedures with `executeStoredProcedure()`” on page 221.

Return values

None.

Exceptions

`CwDBSQLException` – If a database error occurs.

Notes

The `executeStoredProcedure()` method sends a call to the specified *storedProcedure* to the database associated with the current connection. This method sends the stored-procedure call as a prepared SQL statement; that is, the first time it executes, this stored-procedure call is sent as a string to the database, which compiles the string into an executable form (called a prepared statement), executes the SQL statement, and returns this prepared statement to `executeStoredProcedure()`. The `executeStoredProcedure()` method saves this prepared statement in memory.

Important: Before executing a stored procedure with `executeStoredProcedure()`, you must create a `CwDBConnection` object with the `getDBConnection()` method from the `BaseDLM` class.

To handle any data that the stored procedure returns, use the `hasMoreRows()` and `nextRow()` methods.

You can also use the `executeSQL()` or `executePreparedSQL()` method to execute a stored procedure as long as this stored procedure does *not* contain OUT parameters. If the stored procedure uses OUT parameters, you *must* use `executeStoredProcedure()` to execute it. Unlike with `executeSQL()` or `executePreparedSQL()`, you do not have to pass in the full SQL statement to execute the stored procedure. With `executeStoredProcedure()`, you need to pass in only the name of the stored procedure and a `Vector` parameter array of `CwDBStoredProcedureParam` objects. The `executeStoredProcedure()` method can determine the number of parameters from the `storedProcParameters` array and builds the calling statement for the stored procedure.

See also

"Calling stored procedures with `executeStoredProcedure()`" on page 221

"`executePreparedSQL()`" on page 401, `executeSQL()`, `getDBConnection()`, `hasMoreRows()`, `nextRow()`

getUpdateCount()

Returns the number of rows affected by the last write operation to the database.

Syntax

```
int getUpdateCount()
```

Parameters

None.

Return values

Returns an `int` representing the number of rows affected by the last write operation.

Exceptions

`CwDBConnectionException` – If a database error occurs.

Notes

The `getUpdateCount()` method indicates how many rows have been modified by the most recent update operation in the database associated with the current connection. This method is useful after you send an `UPDATE` or `INSERT` statement to the database and you want to determine the number of rows that the SQL statement has affected.

Important: Before using this method, you must create a `CwDBConnection` object with the `getDBConnection()` method from the `BaseDLM` class and send a query that updates the database with either the `executeSQL()` or `executePreparedSQL()` method from the `CwDBConnection` class.

See also

"executePreparedSQL()" on page 401, executeSQL(), getDBConnection()

hasMoreRows()

Determines whether the query result has more rows to process.

Syntax

```
boolean hasMoreRows()
```

Parameters

None.

Return values

Returns true if more rows exist.

Exceptions

CwSQLException – If a database error occurs.

Notes

The hasMoreRows() method determines whether the query result associated with the current connection has more rows to be processed. Use this method to retrieve results from a query that returns data. Such queries include a SELECT statement and a stored procedure. Only one query can be associated with the connection at a time. Therefore, if you execute another query before hasMoreRows() returns false, you lose the data from the initial query.

See also

"Executing static queries that return data (SELECT)" on page 214

"executePreparedSQL()" on page 401, executeSQL(), nextRow()

inTransaction()

Determines whether a transaction is in progress in the current connection.

Syntax

```
boolean inTransaction()
```

Parameters

None.

Return values

Returns true if a transaction is currently active in current connection; returns false otherwise.

Exceptions

CwDBConnectionException – If a database error occurs.

Notes

The `inTransaction()` method returns a boolean value that indicates whether the current connection has an active transaction; that is, a transaction that has been started but not ended.

Important: Before beginning a transaction, you must create a `CwDBConnection` object with the `getDBConnection()` method from the `BaseDLM` class.

See also

"Managing the transaction" on page 224

"`beginTransaction()`" on page 399, "`commit()`" on page 400, `getDBConnection()`, `rollback()`

`isActive()`

Determines whether the current connection is active.

Syntax

```
boolean isActive()
```

Parameters

None.

Return values

Returns `true` if the current connection is active; returns `false` if this connection has been released.

Exceptions

None.

See also

`getDBConnection()`, `release()`

`nextRow()`

Retrieves the next row from the query result.

Syntax

```
Vector nextRow()
```

Parameters

None.

Return values

Returns the next row of the query result as a `Vector` object.

Exceptions

`CwDBSQLException` – If a database error occurs.

Notes

The `nextRow()` method returns one row of data from the query result associated with the current connection. Use this method to retrieve results from a query that returns data. Such queries include a `SELECT` statement and a stored procedure. Only one query can be associated with the connection at a time. Therefore, if you execute another query before `nextRow()` returns the last row of data, you lose the query result from the initial query.

See also

“Executing static queries that return data (SELECT)” on page 214

`hasMoreRows()`, `executePreparedSQL()` on page 401, `executeSQL()`, `executeStoredProcedure()`

release()

Releases use of the current connection, returning it to its connection pool.

Syntax

```
void release()
```

Parameters

None.

Return values

None.

Exceptions

`CwDBConnectionException`

Notes

The `release()` method explicitly releases use of the current connection by the collaboration object. Once released, the connection returns to its connection pool, where it is available for other components (maps or collaborations) that require a connection to the associated database. If you do not explicitly release a connection, the collaboration object implicitly releases it at the end of the current collaboration run. Therefore, you *cannot* save a connection in a static variable and reuse it.

Attention: Do *not* use the `release()` method if a transaction is currently active. With implicit transaction bracketing, ICS does not end the database transaction until it determines the success or failure of the collaboration. Therefore, use of this method on a connection that uses implicit transaction bracketing results in a `CwDBTransactionException` exception. If you do not handle this exception explicitly, it also results in an automatic rollback of the active transaction. You can use the `inTransaction()` method to determine whether a transaction is active.

See also

“Releasing a connection” on page 228

`getDBConnection()`, `inTransaction()`, `isActive()`

rollback()

Rolls back the active transaction associated with the current connection.

Syntax

```
void rollback()
```

Parameters

None.

Return values

None.

Exceptions

CwDBTransactionException – If

Notes

The `rollback()` method ends the active transaction by rolling back any changes made to the database associated with the current connection. The `beginTransaction()`, `commit()` and `rollback()` methods together provide management of transaction boundaries for an explicit transaction. This transaction contains SQL queries, which include the SQL statements INSERT, DELETE, or UPDATE, and a stored procedure that includes one of these SQL statements. If the roll back fails, `rollback()` throws the `CwDBTransactionException` exception and logs an error.

Important: Only use `rollback()` if the connection uses explicit transaction bracketing. If the connection uses implicit transaction bracketing, use of `rollback()` results in a `CwDBTransactionException` exception. If you do not end an explicit transaction with `rollback()` (or `commit()`) before the connection is released, InterChange Server implicitly ends the transaction based on the success of the collaboration. If the collaboration is successful, ICS commits this database transaction. If the collaboration is *not* successful, ICS implicitly rolls back the database transaction. Regardless of the success of the collaboration, ICS logs a warning.

Before beginning an explicit transaction, you must create a `CwDBConnection` object with the `getDBConnection()` method from the `BaseCollaboration` class. Make sure that this connection uses explicit transaction bracketing

Examples

For an example of managing a transaction with `rollback()`, see the example in “Transaction scope with explicit transaction bracketing” on page 226.

See also

“Managing the transaction” on page 224

`beginTransaction()` on page 399, `commit()` on page 400, `getDBConnection()`, `inTransaction()`

Chapter 26. CwDBStoredProcedureParam class

A CwDBStoredProcedureParam object describes a single parameter for a stored procedure. Table 86 summarizes the methods in the CwDBStoredProcedureParam class.

Table 86. CwDBStoredProcedureParam method summary

Method	Description	Page
CwDBStoredProcedureParam()	Constructs a new instance of CwDBStoredProcedureParam that holds argument information for the parameter of a stored procedure.	411
getParamType()	Retrieves the in/out type of the current stored-procedure parameter as an integer constant.	412
getValue()	Retrieves the value of the current stored-procedure parameter.	413

CwDBStoredProcedureParam()

Constructs a new instance of CwDBStoredProcedureParam that holds argument information for the parameter of a stored procedure.

Syntax

```
CwDBStoredProcedureParam(int paramType, String paramValue);  
  
CwDBStoredProcedureParam(int paramType, int paramValue);  
CwDBStoredProcedureParam(int paramType, Integer paramValue);  
CwDBStoredProcedureParam(int paramType, Long paramValue);  
  
CwDBStoredProcedureParam(int paramType, double paramValue);  
CwDBStoredProcedureParam(int paramType, Double paramValue);  
CwDBStoredProcedureParam(int paramType, float paramValue);  
CwDBStoredProcedureParam(int paramType, Float paramValue);  
CwDBStoredProcedureParam(int paramType, BigDecimal paramValue);  
  
CwDBStoredProcedureParam(int paramType, boolean paramValue);  
CwDBStoredProcedureParam(int paramType, Boolean paramValue);  
  
CwDBStoredProcedureParam(int paramType, java.sql.Date paramValue);  
CwDBStoredProcedureParam(int paramType, java.sql.Time paramValue);  
CwDBStoredProcedureParam(int paramType, java.sql.Timestamp paramValue);  
  
CwDBStoredProcedureParam(int paramType, java.sql.Blob paramValue);  
CwDBStoredProcedureParam(int paramType, java.sql.Clob paramValue);  
  
CwDBStoredProcedureParam(int paramType, byte[] paramValue);  
CwDBStoredProcedureParam(int paramType, Array paramValue);  
CwDBStoredProcedureParam(int paramType, Struct paramValue);
```

Parameters

<i>paramType</i>	The in/out parameter type of the associated stored-procedure parameter.
<i>paramValue</i>	The argument value to send to the stored procedure. This value is one of the following Java data types

Return values

Returns a new `CwDBStoredProcedureParam` object to hold the argument information for one argument in the declaration of the stored procedure.

Exceptions

None.

Notes

The `CwDBStoredProcedureParam()` constructor creates a `CwDBStoredProcedureParam` instance to describe one parameter for a stored procedure. Parameter information includes the following:

- The parameter's in/out type
The constructor's first argument initializes this in/out parameter type. For a list of valid in/out parameter types, see Table 87 on page 413.
- The parameter value
The constructor's second argument initializes this parameter value. The `CwDBStoredProcedureParam` class provides one form of its constructor for each of the parameter-value data types it supports. For a list of the mappings between Java data types and JDBC data types for stored-procedure parameters, see Table 62 on page 224.

You provide a Java `Vector` of stored-procedure parameters to the `executeStoredProcedure()` method, which creates a stored-procedure call from a stored-procedure name and the parameter vector, and sends this call to the database associated with the current connection.

See also

"Calling stored procedures with `executeStoredProcedure()`" on page 221

`executeStoredProcedure()`

`getParamType()`

Retrieves the in/out type of the current stored-procedure parameter as an integer constant.

Syntax

```
int getParamType()
```

Parameters

None.

Return values

Returns the in/out type of the associated `CwDBStoredProcedureParam` parameter.

Exceptions

None.

Notes

The `getType()` method returns the in/out parameter type of the current stored-procedure parameter. The in/out parameter type indicates how the stored procedure uses the parameter. The `CwDBStoredProcedureParam` class represents each in/out type as a constant, as Table 87 shows.

Table 87. Parameter In/Out Types

Parameter In/Out Type	Description	In/Out Type Constant
IN parameter	An IN parameter is <i>input only</i> ; that is, the stored procedure accepts its value as input but does <i>not</i> use the parameter to return a value.	PARAM_IN
OUT parameter	An OUT parameter is <i>output only</i> ; that is, the stored procedure does <i>not</i> read its value as input but does use the parameter to return a value.	PARAM_OUT
INOUT parameter	An INOUT parameter is <i>input and output</i> ; that is, the stored procedure accepts its value as input and also uses the parameter to return a value.	PARAM_INOUT

See also

“Calling stored procedures with `executeStoredProcedure()`” on page 221

`CwDBStoredProcedureParam()`, `getValue()`

getValue()

Retrieves the value of the current stored-procedure parameter.

Syntax

```
Object getValue()
```

Parameters

None.

Return Values

Returns the value of the associated `CwDBStoredProcedureParam` parameter as a Java Object.

Exceptions

None.

Notes

The `getParamValue()` method returns the parameter value as a Java Object (such as Integer, Double, or String). If the value returned to an OUT parameter is the JDBC NULL, `getParamValue()` returns the null constant.

See Also

`CwDBStoredProcedureParam()`, `getParamType()`

Chapter 27. CxExecutionContext class

The methods documented in this chapter operate on the global execution context, which is a holder for user-accessible context information that is associated with a given flow. It is represented by the WebSphere InterChange Server-defined class, `CxExecutionContext`. Currently, only the map-specific execution information is surfaced as the map execution context. In the code of a map, Map Designer automatically declares a special variable to access the map execution context, `cxExecCtx`. However, when you call a map from within a collaboration, you must instantiate your own global execution context and map execution context. For more information, see “Calling a native map” on page 199.

Note: For more information on the map execution context, see the *Map Development Guide*.

Table 88 summarizes the methods of the `CxExecutionContext` class.

Table 88. *CxExecutionContext* method summary

Method	Description	Page
<code>CxExecutionContext()</code>	Constructs a new instance of a global execution context.	415
<code>getContext()</code>	Retrieve the specified execution context from the global execution context.	416
<code>setContext()</code>	Sets a particular execution context to be part of the global execution context.	416

Static constants

The `CxExecutionContext` class defines the static constants that Table 89 shows.

Table 89. *Static constants defined in the CxExecutionContext class*

Constant name	Meaning
<code>MAPCONTEXT</code>	A string constant to indicate that the execution context is map-specific

CxExecutionContext()

Constructs a new instance of a global execution context.

Syntax

```
CxExecutionContext()
```

Parameters

None

Return values

Returns the new instance of the global execution context.

Notes

The `CxExecutionContext()` constructor returns a global execution context, which is needed to hold the map execution context before calling a map from a collaboration.

See also

“Calling a native map” on page 199

getContext()

Retrieve the specified execution context from the global execution context.

Syntax

```
Object getContext(String contextName)
```

Parameters

contextName The name of a execution context to obtain from the global execution context.

Return values

Returns an instance of the specified execution context.

Examples

The following example obtains a map execution context from a global execution context.

```
(MapExeContext) mapExeContext = globalExeContext.getContext(  
    CxExecutionContext.MAPCONTEXT);
```

See also

“Calling a native map” on page 199

setContext()

Sets a particular execution context to be part of the global execution context.

Syntax

```
void setContext(String contextName, Object context)
```

Parameters

contextName The name of the specific execution context to assign to the global execution context. Currently, `MAPCONTEXT` is the only valid value.

context An object that contains the information for the execution context. For map execution contexts, this `Object` is of type `MapExeContext`.

Return values

None

Examples

The following example saves a map execution context into a global execution context:

```
globalExeContext.setContext(CxExecutionContext.MAPCONTEXT,  
    mapExeContext);
```

See also

“Calling a native map” on page 199

Chapter 28. CxBidiEngine class

The CxBidiEngine class provides methods for transforming business objects and strings from one bidirectional format to the other.

Table 90 summarizes the methods in the CxBidiEngine class.

Table 90. CxBidiEngine method summary

Method	Description	Page
BiDiB0Transformation()	Transforms BusinessObject type business objects from one bidirectional format to the other format.	419
BiDiBusObjTransformation()	Transforms BusObj type business objects from one bidirectional format to the other format.	420
BiDiStringTransformation()	Transforms strings from one bidirectional format to the other.	421

BiDiB0Transformation()

The BiDiTransformation() method transforms BusinessObject type business objects from one bidirectional format to the other format. Use this method when you develop controllers, connectors and maps.

Syntax

```
BusinessObject BiDiB0Transformation(BusinessObject boIn, String formatIn,  
String formatOut, boolean replace)
```

Parameters

<i>boIn</i>	The business object to transform. The object must be of the BusinessObject type.
<i>formatIn</i>	A string that represents the bidirectional format of the input business object content. See Table 91 on page 420 for the valid values of this string. If this parameter is null, the method defaults to the standard Windows bidirectional format.
<i>formatOut</i>	A string that represents the bidirectional format of the output business object content. See Table 91 on page 420 for the valid values of this string. If this parameter is null, the method defaults to the standard Windows bidirectional format.
<i>replace</i>	A value that specifies whether the input business object is to be replaced. The valid value is either true or false.

Return values

The return value is a transformed business object. If the method is unsuccessful, it returns a null value.

Exceptions

None.

Examples

See the example in “BiDiStringTransformation()” on page 421.

BiDiBusObjTransformation()

The BiDiBusObjTransformation() method transforms BusObj type business objects from one bidirectional format to the other. Use this method within collaborations.

Syntax

```
BusObj BiDiBusObjTransformation(BusObj busObjIn, String formatIn,  
String formatOut, boolean replace)
```

Parameters

<i>busObjIn</i>	The business object to transform. The object must be of the BusObj type.
<i>formatIn</i>	A string that represents the bidirectional format of the input business object content. See Table 91 for the valid values of this string. If this parameter is null, the method defaults to the standard Windows bidirectional format.
<i>formatOut</i>	A string that represents the bidirectional format of the output business object content. See Table 91 for the valid values of this string. If this parameter is null, the method defaults to the standard Windows bidirectional format.
<i>replace</i>	A value that specifies whether the input business object is to be replaced. The valid value is either true or false.

Table 91. Values for format strings

Letter position	Purpose	Values	Description	Default
1	Type	I	Implicit (Logical)	I
		V	Visual	
2	Direction	L	Left to Right	L
		R	Right to Left	
3	Symmetric swapping	Y	Symmetric swapping is on	Y
		N	Symmetric swapping is off	
4	Shaping	Y	Text is shaped	N
		N	Text is not shaped	
5	Numeric shaping	H	Hindi	N
		C	Contextual	
		N	Nominal	

Return values

The return value is a transformed business object. If the method is unsuccessful, it returns a null value.

Exceptions

None.

Examples

This example transforms InputBOBusObj from the standard Windows bidirectional format to the visual bidirectional format.

```
BusObj dummyBusObj = null;
dummyBusObj = CwBidiEngine.BiDiBusObjTransformation(
    InputBOBusObj,
    "ILYNN",
    "VLYNN", true);
```

BiDiStringTransformation()

The BiDiStringTransformation() method transforms strings from one bidirectional format to the other.

Syntax

```
BiDiStringTransformation(String strIn, String formatIn, String formatOut
```

Parameters

- strIn* The string to transform.
- formatIn* A string that represents the bidirectional format of the input business object content. See Table 92 for the valid values of this string. If this parameter is null, the method defaults to the standard Windows bidirectional format.
- formatOut* A string that represents the bidirectional format of the output business object content. See Table 92 for the valid values of this string. If this parameter is null, the method defaults to the standard Windows bidirectional format

Table 92. Values for format strings

Letter position	Purpose	Values	Description	Default
1	Type	I	Implicit (Logical)	I
		V	Visual	
2	Direction	L	Left to Right	L
		R	Right to Left	
3	Symmetric swapping	Y	Symmetric swapping is on	Y
		N	Symmetric swapping is off	
4	Shaping	Y	Text is shaped	N
		N	Text is not shaped	
5	Numeric shaping	H	Hindi	N
		C	Contextual	
		N	Nominal	

Return values

The return value is a transformed string.

Exceptions

None.

Examples

The following example applies the `BiDiStringTransformation()` method to the attribute values of a business object.

```
for (int i = 0; i < bo.getAttrCount(); i++) {
    intAttrType = bo.getAttributeType(i);
    Object attrValue = bo.getAttrValue(i);
    String attrName = bo.getAttrName(i);

    if (attrValue != null {
        // We handle only String or Long Text Attribute and not
        // the ObjectEventId attribute
        if (((attrType == CxObjectAttrType.STRING)
            || (attrType == CxObjectAttrType.LONGTEXT))
            && !(attrName.equals(OBJECT_EVENT_ID))) {
            String strOut = BiDiStringTransformation(attrValue.toString(),
                bo.setAttrValue(i, strOut);
        } else if (attrType == CxObjectAttrType.OBJECT) {
            CxObjectAttr attrDesc = bo.getAttrDesc(i);
            if (attrDesc.getCardinality().equals(CxObjectAttr.CARD_Single)) {
                BiDiTransformation((BusinessObject) attrValue, "ILYNN",
                    "VLYNN",
                    true);
            } else {
                // multiple cardinality
                CxObjectContainer cont = (CxObjectContainer) attrValue;
                int objCount = cont.getObjectCount();
                for (int j = 0; j < objCount; j++) {
                    BiDiBTransformation((BusinessObject) (cont.getObject(j)),
                        "ILYNN",
                        "VLYNN",
                        true);
                }
            }
        }
    }
}
```

Chapter 29. CollaborationException class

The methods documented in this chapter operate on objects of the `CollaborationException` class. These objects represent collaboration exceptions. Exceptions might occur during the execution of a collaboration object. The scenario can catch and handle these exceptions. There are two categories of exceptions that a collaboration can handle:

- Business process exceptions

Business process exceptions arise from code that uses the collaboration API methods. For example, a business process exception can occur when the scenario sets the value of a business object attribute, sends a request to a connector, and so on.

- Native Java exceptions

Java exceptions result from your own code that uses native Java methods. The collaboration runtime environment catches and handles the Java exceptions arising from its own code.

Table 93 lists the methods that this chapter describes.

Table 93. CollaborationException method summary

Method	Description	Page
<code>getMessage()</code>	Retrieve the message text from the current exception.	423
<code>getMsgNumber()</code>	Retrieve the message number of the text associated with the current exception.	424
<code>getSubType()</code>	Retrieve the subtype of an exception.	424
<code>getType()</code>	Retrieve the collaboration exception type.	425
<code>toString()</code>	Write exception information to a string.	426

getMessage()

Retrieve the message text from the exception object.

Syntax

```
String getMessage()
```

Return values

A `String` that contains the message associated with the exception object.

Notes

The `getMessage()` method is useful for extracting the exception text from the `currentException` system variable. This exception text can be included in a call to `raiseException()` to ensure that the reason for the exception is raised up to the next higher level of execution.

Note: You can use the `toString()` method to retrieve the exception type and exception text from the current exception as a formatted string.

getMsgNumber()

Retrieve the message number for the message associated with the exception object.

Syntax

```
int getMsgNumber()
```

Return values

The integer (int) message number associated with the current exception's message. If the exception's message is not from a message file, this method returns zero (0).

Notes

The `getMsgNumber()` method is useful for obtaining the message number associated with an exception's message. You can pass this message number to a call to `raiseException()` or `logError()`.

getSubType()

Retrieve the exception subtype from the exception object.

Syntax

```
String getSubType()
```

Return values

A `String` that contains the exception subtype for the current exception. For more information on valid exception subtypes, see the Notes section.

Notes

The `getSubType()` method retrieves the exception subtype for the current exception. For exceptions whose exception type does not adequately identify the cause of the exception, the exception subtype can provide more information. The following exception types most commonly use exception subtypes:

- `JavaException`

The collaboration runtime environment catches Java exceptions and wraps them in a collaboration exception with an associated type of Java exception. A collaboration can use `getSubType()` on the collaboration exception to retrieve the original type of the Java exception (that is, the class name of the captured Java exception). However, this should normally not be necessary.

- `ServiceCallException`

The `ServiceCallException` exception type occurs if any failure results from a service call. To develop more robust collaborations, you can use the exception subtype to determine the cause of the service-call failure. The valid exception subtypes include:

<code>AppTimeOut</code>	Aconnector was unable to complete communication with its application.
<code>AppLogOnFailure</code>	Aconnector was unable to log in to the application.
<code>AppRetrieveByContentFailed</code>	A Retrieve by non-key values, performed on the application, was not able to find any match.
<code>AppMultipleHits</code>	An application found and retrieved more than one entity in response to a Retrieve request.

AppBusObjDoesNotExist	A Retrieve operation was performed on the application, but the entity that the business object represents does not exist in the application database.
AppRequestNotYetSent	In the case of a parallel connector agent, the request was queued up in the agent master but never got dispatched to the application; therefore, you can resend the request. For more information, see “Unsent service call requests” on page 181.
ServiceCallTransportException	There was an error in the transport, and it cannot be determined with certainty whether the request reached the application. For more information, see “Handling runtime transport-related exceptions” on page 179.
AppUnknown	Any type of error that is not one of the other subtypes. If this exception subtype is present, the application operation requested in the service call might be finished or not finished.

For more information, see “Handling particular service-call exceptions” on page 178.

Important: The AppTimeout, AppLogOnFailure, AppRetrieveByContent, AppMultipleHits, and AppUnknown exception subtypes correspond to outcome-status values that an adapter can return to indicate the cause of failure. Older adapters might not support all of the corresponding outcome-status values. Make sure you rigorously test any adapters that are bound to your collaboration with the Test Connector tool to determine the actual outcome-status values they return.

Examples

This section provides examples of retrieve exception subtypes for the following exception types:

- **JavaException**

The code in the following example retrieves a Java exception thrown by a mathematical function.

```
//
// If the preceding division operation threw an exception,
// set the result to 0.
//
if (currentException.getType().equals("JavaException"))
{
    String subType = currentException.getSubType();
    logWarning("Caught exception " + subType +
        ". Setting result to 0.");
    result = 0;
}
```

- **ServiceCallException**

For examples of how to handle two of the ServiceCallException subtypes, see the specified sections:

- For AppRequestNotYetSent, see “Unsent service call requests” on page 181.
- For ServiceCallTransportException, see “Handling runtime transport-related exceptions” on page 179.

getType()

Retrieve the collaboration exception type from the exception object.

Syntax

```
String getType()
```

Return values

A `String` that contains the exception type for the current exception. Compare this string value with one of the following exception-type static variables:

<code>AnyException</code>	Any type of exception. If there are two exception links, one that tests for a specific type of exception and one that tests for <code>AnyException</code> , the link that tests for the specific type of exception is checked first. If the current exception does not match the specific exception, the link that tests for <code>AnyException</code> is processed next.
<code>AttributeException</code>	Attribute access problem. For example, the collaboration called <code>getDouble()</code> on a <code>String</code> attribute or called <code>getString()</code> on a nonexistent attribute.
<code>JavaException</code>	Problem with Java code in the collaboration logic.
<code>ObjectException</code>	Business object passed to a method was invalid or a null object was accessed.
<code>OperationException</code>	Service call was improperly set up and could not be sent.
<code>ServiceCallException</code>	Service call failed. For example, a connector or application is unavailable.
<code>SystemException</code>	WebSphere InterChange Server internal error.
<code>TransactionException</code>	Error related to the transactional behavior of a transactional collaboration. For example, rollback failed or the collaboration could not apply compensation.

Notes

The `getType()` method retrieves the exception type from the current exception. The exception type is a `String` that identifies the cause of the exception.

Examples

The following example retrieves the collaboration exception type and uses it in a call to the `raiseException()` method.

```
String problem currentException.getType();
raiseException(problem, 1234);
```

toString()

Format exception information, including the exception type and text, to a string.

Syntax

```
String toString()
```

Parameters

exception The variable holding an exception object.

Notes

The `toString()` method formats the exception information for the current exception as follows:

```
exceptionType: messageText
```

In the line above, *exceptionType* is the exception object's exception type and *messageText* is its exception text.

Note: You can use the `getMessage()` method to retrieve only the exception text from the current exception.

Examples

The following example writes the current exception information to the `String` variable `newmessage`:

```
String newmessage = currentException.toString();
```

Deprecated methods

Table 94 lists the deprecated methods that were available in prerelease versions of Process Designer. If you have not used Process Designer previously, ignore this section. For an explanation of deprecation, see “Deprecated method” on page 384.

Table 94. Deprecated methods, exception class

Former method	Replacement
<code>getText()</code>	<code>toString()</code>

Chapter 30. EventManagement package

The classes in this chapter instantiate objects in the EventManagement package. These objects are arrays used by methods in the BaseCollaboration class.

Table 95 lists the classes within the EventManagement package that this chapter describes.

Table 95. Classes in the EventManagement package

Class	Description	Page
EventKeyAttrDef()	Constructs a new instance of the EventKeyAttrDef class.	429
EventQueryDef()	Constructs a new instance of the EventQueryDef class.	431
FailedEventInfo()	Creates a new instance of the FailedEventInfo class.	433

EventKeyAttrDef()

EventKeyAttrDef()

Constructs a new instance of the EventKeyAttrDef array.

Syntax

```
EventKeyAttrDef()
```

Parameters

None.

Return values

Returns an array with the following structure:

keyName *KeyName* is the attribute name.

keyValue *KeyValue* is the attribute value.

Examples

```
myEventKeys new EventKeyAttrDef();
```

See also

“FailedEventInfo()” on page 433

“queryFailedEvents()” on page 355

EventQueryDef()

Static constants

The EventQueryDef() class defines the static constants shown in Table 96.

Table 96. Static constants in the EventQueryDef() class

Constant	Meaning
STATUS_ANY_UNRESOLVED	Events with any unresolved status.
STATUS_FAILED	Events with failed status.
STATUS_DEFERRED_RECOVERY	Events with deferred recovery status.
STATUS_SERVICECALL_IN_TRANSIT	Events with service calls in process.
STATUS_DELIVER_POSSIBLE_DUPLICATE	Events that could be duplicates.
STATUS_SERVICECALL_WAITING	Events waiting for a service call to complete.
STATUS_USER_GENERATED	Events saved with the saveFailedEvent() method.

EventQueryDef()

Instantiates an array of class EventQueryDef.

Syntax:

```
EventQueryDef()
```

Parameters: None.

Examples:

```
myQueryOptions new EventQueryDef();
```

Return values: An array with structure:

nameConnector Name of the connector of interest.

nameCollaboration
Name of collaboration of interest.

nameBusObj Name of the business object of interest.

verb name of the verb of interest. If left null, all verbs will be selected.

See also:

“queryFailedEvents()” on page 355

FailedEventInfo()

Static constants

The FailedEventInfo() class defines the static constants shown in Table 97.

Table 97. Static constants in the FailedEventInfo() class.

Constant	Meaning
RESUBMIT_ORIGINAL_BUSOBJ	Resubmit the failed event with the original business object.
RESUBMIT_NEW_BUSOBJ	Resubmit the failed event with a new business object.

FailedEventInfo()

Instantiates an array of FailedEventInfo.

Syntax

```
FailedEventInfo()
```

Parameters

None

Examples

```
myFailedEventInfo new FailedEventInfo();
```

Return values

Returns an array with the following structure:

nameOwner *NameOwner* is the name of the collaboration to which this event belongs.

nameConnector *NameConnector* is the name of the connector to which this event belongs.

nameBusObj *NameBusObj* is the name of the business object involved in this event.

nameVerb *NameVerb* is the verb executing during this event.

strTime *StrTime* is the time of the event.

strMessage *strMessage* is a message associated with the failing event.

wipIndex *wipIndex* is an integer that indicates the event's position.

EventKeyAttrDef[]

An array as defined in "EventKeyAttrDef()" on page 430.

nkeys *NKeys* is the number of key attributes in the **EventKeyAttrDef[]** array.

eventStatus *EventStatus* is an integer that indicates the status of the event which can be one of the static constants defined by the EventQueryDef() class in "Static constants" on page 431.

expirationTime *ExpirationTime* is the time at which the event occurred.

scenarioName *ScenarioName* is the name of the scenario occurring at the time of the event.

scenarioState *ScenarioState* is the completion state of the running scenario.

See also

"EventKeyAttrDef()" on page 429

"EventKeyAttrDef()" on page 429

"queryFailedEvents()" on page 355

Chapter 31. Filter class

The methods documented in this chapter operate on objects of the Filter class. Data filtering is a common task in collaboration processing; these methods filter the data contained in specified attributes of a business object. The results can then be used to determine whether the collaboration needs to synchronize a business object with specific data.

Table 98 lists the methods that this chapter describes.

Table 98. Filter method summary

Method	Description	Page
Filter()	Constructs a new instance of the Filter class.	436
filterExcludes()	Determines whether the given attribute value is equal to the given exclusion value or values.	437
filterIncludes()	Determines whether the given attribute value is equal to the given inclusion value or values.	438
recurseFilter()	Determines whether the given attribute value is equal to the given inclusion or exclusion value or values.	439
recursePreReqs()	Recursively finds the vector position of a specified business object type within a given vector of unique business objects.	440

Filter()

Constructs a new instance of Filter.

Syntax

```
Filter(BaseCollaboration baseCollab)
```

Parameters

baseCollab

Specifies the current collaboration instance.

Return values

Returns a newly instantiated Filter object.

filterExcludes()

Determines whether the specified attribute value is equal to that of the exclusion values.

Syntax

```
boolean filterExcludes(String FilterAttributeValue,  
                       String ExcludeValues)
```

Parameters

FilterAttributeValue

The value of the attribute being filtered.

ExcludeValues

The values the collaboration uses as a filter to prevent synchronization of the business object.

Return values

Returns `False` if the value of *FilterAttributeValue* matches one of the values listed in *ExcludeValues*. Otherwise, the method returns `True`.

filterIncludes()

Determines whether the specified attribute value is equal to that of the inclusion values.

Syntax

```
boolean filterIncludes(String FilterAttributeValue, String IncludeValues)
```

Parameters

FilterAttributeValue

The value of the attribute being filtered.

IncludesValues

The values the collaboration uses as a filter to allow synchronization of the business object.

Return values

Returns True if the value of *FilterAttributeValue* matches one of the values listed in *IncludesValues*. Otherwise, the method returns False.

recurseFilter()

Determines whether the specified attribute value is equal to that of the exclusion or inclusion values.

Syntax

```
boolean recurseFilter(BusObj busObj,  
                     String filterAttribute,  
                     boolean stopOnFail,  
                     String includeValues,  
                     String excludeValues)
```

Parameters

busObj The business object instance on which to filter.

filterAttribute

The name of the business object attribute used when comparing values specified by *includeValues* and *excludeValues*. The collaboration compares the value in the filter attribute against the specified inclusion or exclusion values to either prevent or enable synchronization of the business object.

stopOnFail

Specifies how to handle the value of the *filterAttribute* attribute if it does not meet the filtering criteria.

includesValues

The values the collaboration uses as a filter to allow synchronization of the business object.

excludesValues

The values the collaboration uses as a filter to prevent synchronization of the business object.

Return values

Returns True if *filterAttribute* contains a value specified in *includesValues* or a value not specified in *excludesValues*. Otherwise, the method returns False.

Exceptions

CollaborationException—This exception is thrown if the attribute value of *filterAttribute* is not specified as an included value but rather as an excluded value and the *stopOnFail* parameter is set to True.

recursePreReqs()

Recursively finds the vector position of a specified business object type within a given vector of unique business objects.

Syntax

```
int recursePreReqs(String Type, Vector busObjs)
```

Parameters

Type The type of business object to search for in the *busObj* vector.

busObjs
The vector of business objects of unique business object types.

Return values

Returns the position in the *busObjs* vector that contains the business object specified by *Type*.

Exceptions

CollaborationException—This exception is thrown if the collaboration configuration property `PREQ_Type` is missing.

Chapter 32. Globals class

The methods documented in this chapter operate on objects of the Globals class.

The Globals class maintains a global hash table to support asynchronous event processing. In previous releases of InterChange Server, a collaboration communicated with other collaborations and connectors exclusively through synchronous service calls. With the introduction of the Globals class, collaborations can now send or receive a communication from another collaboration or connector, wait for a specified amount of time for the appropriate response, and then continue processing the event in the same thread.

Table 99 lists the methods in the Globals class.

Table 99. Globals class method summary

Method	Description	Page
Globals()	Constructs a new instance of the Globals class.	442
callMap()	Provides a wrapper around the DtpMapService.runMap API to make it easier to call a map from within a collaboration.	443

Globals()

Constructs a new instance of the Globals class.

Syntax

```
Globals(BaseCollaboration baseCollab)
```

Parameters

baseCollab

Specifies the current collaboration instance.

Return values

A newly instantiated Globals object.

callMap()

Provides a wrapper around the `DtpMapService.runMap` API to make it easier to call a map from within a collaboration.

Syntax

```
BusObj callMap(String mapName, BusObj srcBusObj)
```

Parameters

mapName

Specifies the name of the map you want to run.

srcBusObj

Specifies the source business object for the map.

Return values

Returns the destination business object for the map specified by *mapName*.

Exceptions

`CollaborationException`—Thrown if an error occurs while attempting to run the *mapName* map.

Chapter 33. SmartCollabService class

The methods documented in this chapter operate on objects of the SmartCollabService class. This class provides a set of methods to simplify the splitting, merging, and aggregation of array attributes in a business object.

Table 100 lists the methods provided in the SmartCollabService class.

Table 100. SmartCollabService method summary

Method	Description	Page
SmartCollabService()	Constructs a new instance of the SmartCollabService class.	445
doAgg()	Aggregates like container attributes into one container attribute, based on user-specified criteria and attributes.	446
doMergeHash()	Takes a collection of business objects and groups them under a new parent business object specified by the split level. The business objects are grouped by like content specified in the key attribute or attributes.	446
doRecursiveAgg()	Recursively aggregates hierarchical like container attributes into one container attribute, based on user-specified criteria and attributes.	447
doRecursiveSplit()	Retrieves the container business objects from a particular level of a business object hierarchy, and optionally returns within the top-level business object.	447
getKeyValues()	Calculates the key value for the business object to be used by a hash table, based on the comma-separated value specified in the key attribute or attributes.	448
merge()	Merges a collection of business objects under one top-level business object.	448
split()	Splits a business object into container business objects, as specified by the split level.	449

SmartCollabService()

Constructs a new instance of SmartCollabService.

Syntax

```
SmartCollabService()  
SmartCollabService(com.crossworlds.BaseCollaboration baseCollab)
```

Parameters

baseCollab

Specifies the current collaboration instance.

Return values

Returns a newly instantiated SmartCollabService object.

doAgg()

Aggregates like container attributes into a single container attribute according to user-specified criteria and the list of attributes to be aggregated.

Syntax

```
BusObj doAgg(BusObj inBusObj,  
             String Level,  
             String KeyAttr,  
             String Attr)
```

Parameters

inBusObj

Specifies the business object to be aggregated.

Level

Specifies the aggregation level, which comprises the type of the business object (its business object definition) and the name of the array attribute to aggregate. Names are delimited by a period (.).

KeyAttr

Specifies the key attributes of the business object used for aggregation.

Attr

Specifies the attributes to be aggregated. Multiple names are delimited by a comma (,).

Return values

Returns the aggregated business object.

Exceptions

CollaborationException—Thrown if an error occurs during aggregation of the business object attributes.

doMergeHash()

Groups a collection of business objects under a new parent business object, as specified by the split level. Business objects are grouped by like content in the key attribute or attributes.

Syntax

```
java.util.Vector doMergeHash(java.util.Vector BusObj,  
                             String Level,  
                             String KeyAttr)
```

Parameters

BusObj

Specifies a vector that contains the collection of business objects to be merged.

Level

Specifies the merge level, which comprises the type of the parent business object (its business object definition) and the attribute designated to hold the child business object. Names are delimited by a period (.).

KeyAttr

Specifies the business object's attributes that are used as merging criteria. Values are delimited by a comma (,).

Return values

Returns the vector of merged business objects.

Exceptions

`CollaborationException`—Thrown if an error occurs while merging the business objects.

doRecursiveAgg()

Recursively aggregates like hierarchical array attributes into a single container attribute according to user-specified criteria and the list of attributes to be summed up.

Syntax

```
BusObj doRecursiveAgg(BusObj inBusObj,  
    String Level,  
    String KeyAttr,  
    String Attr)
```

Parameters

inBusObj

Specifies the business object to be aggregated.

Level

Specifies the aggregation level, which comprises the type of the business object (its business object definition) and the name of the array attribute to aggregate. Names are delimited by a period (.).

KeyAttr

Specifies the key attributes of the business object used for aggregation.

Attr

Specifies the attributes to be aggregated. Multiple names are delimited by a comma (,).

Return values

Returns the aggregated business object.

Exceptions

`CollaborationException`—Thrown if an error occurs during aggregation of the business object attributes.

doRecursiveSplit()

Retrieves the container business objects from a specified level within the business object hierarchy.

Syntax

```
java.util.Vector doRecursiveSplit(BusObj inBusObj,  
    String Level)  
java.util.Vector doRecursiveSplit(BusObj inBusObj,  
    String Level,  
    boolean KeepParents)
```

Parameters

inBusObj

Specifies the top-level business object whose array attribute the method splits.

Level

Specifies the path to the array attribute on which the business object is to be split. Values are delimited by a period (.).

KeepParents

Specifies whether the split business object is returned in the parent business object or as a standalone object. Set the parameter to True to return the split business object within the parent business object.

Return values

Returns a vector of business objects.

Exceptions

CollaborationException—Thrown if an error occurs while splitting the business object.

getKeyValues()

Calculates a business object's key value for use in a hash table.

Syntax

```
String getKeyValues(BusObj inBusObj,  
String KeyAttr)
```

Parameters

inBusObj

Specifies the business object for which the key value is calculated.

KeyAttr

Specifies the attribute name on which the method operates. Multiple values are delimited by a comma (,).

Return values

Returns the key value to be used in a Java hash table.

Exceptions

CollaborationException—Thrown if an error occurs while calculating the business object's key value.

merge()

Merges a collection of business objects under one top-level business object.

Syntax

```
BusObj merge(java.util.Vector BusObjs,  
String Level)  
BusObj merge(java.util.Vector BusObjs,  
String Attr,  
BusObj mergeBusObj)
```


Parameters

BusObjs

Specifies the collection of child business objects to be merged.

Level

Specifies the merge level, which comprises the business object type and the name of its array attribute whose child business objects are going to be merged. Names are delimited by a period (.).

Attr

Specifies the name of the array attribute in the *mergeBusObj* business object in which the child business objects are to be merged.

mergeBusObj

Specifies the top-level business object that is going to hold the merged collection of child business objects.

Return values

Returns the top-level business object (either new or specified) that contains the merged collection of child business objects.

Exceptions

CollaborationException—Thrown if an error occurs while merging business objects.

split()

Splits a business object into the number of container business objects specified by the split level.

Syntax

```
Vector split(BusObj inBusObj,  
            String Attr)
```

Parameters

inBusObj

Specifies the parent-level business object on which the split() method operates.

Attr

Specifies the name of the array attribute on which the business object is to be split.

Return values

Returns a vector of business objects, one business object for each child business object in the parent business object's array attribute.

Exceptions

CollaborationException—Thrown if an error occurs while splitting the business object.

Chapter 34. StateManagement class

The StateManagement class enables you to manage the state of a collaboration and the persistence of business objects. State management and business object persistence are necessary for implementing long-lived business processes.

A collaboration's state is managed by performing save, retrieve, update, and delete operations on the CxCollabState database table. This table contains the following attributes:

- Id
- Verb
- CollabObjName
- BusinessObjectType
- PropDocID
- State
- Retry
- BeginTime
- TimeOut

A business object's persistence is managed by performing save, retrieve, update, and delete operations on the CxCollabStateBO database table. This table has the following attributes:

- CollabObjName
- Verb
- PropDocID
- BusinessObjectType
- BusObj

Table 101 describes the methods in the StateManagement class.

Table 101. StateManagement method summary

Method	Description	Page
beginTransaction()	Marks the beginning of a transaction.	452
commit()	Commits a transaction.	452
deleteBO()	Deletes a persisted business object from the CxCollabStateBO database table.	452
deleteState()	Deletes entries from the CxCollabState database table.	453
persistBO()	Persists a business object in the CxCollabStateBO database table.	453
recoverBO()	Recovers a business object that has been persisted in the CxCollabStateBO database table.	454
releaseDBConnection()	Releases the database connection.	455
resetData()	Resets the value of the boolean variable <i>bTranStarted</i> .	455
retrieveState()	Retrieves the latest value of the retry count stored in the CxCollabState database table.	455

Table 101. StateManagement method summary (continued)

Method	Description	Page
saveState()	Saves the collaboration process parameters in the CxCollabState database table.	456
setDBConnection()	Sets the database connection.	456
StateManagement()	Creates and initializes a StateManagement object.	457
updateBO()	Updates a persisted business object in the CxCollabStateBO database table.	457
updateState()	Updates the retry count value in the CxCollabState database table.	457

beginTransaction()

Marks the beginning of a transaction.

Syntax

```
public void beginTransaction()
```

Exceptions

CwDBConnectionException—This exception is thrown if the StateManagement class is unable to begin the transaction.

commit()

Commits a transaction.

Syntax

```
public void commit()
```

Exceptions

Throws the following exceptions:

- CwDBTransactionException—This exception occurs when the StateManagement class is unable to commit the transaction.
- CwDBConnectionException—This exception occurs when the StateManagement class is unable to begin the transaction.

deleteBO()

Deletes a persisted business object from the CxCollabStateBO database table.

Syntax

```
public void deleteBO(java.lang.String Verb,  
                    java.lang.String PropDocID,  
                    java.lang.String BusinessObjectType)
```

Parameters

Verb Specifies the verb to use. In this case, the value must be Delete.

BusinessObjectType
Specifies the type of the business object you are deleting.

PropDocID

Specifies identifier for the business object you are deleting.

Exceptions

Throws the following exceptions:

- *CwDBSQLException*—Thrown if the *StateManagement* class is unable to execute the SQL query.
- *CollaborationException*—Thrown if the values of the parameters passed are null, or of any other unknown exception occurs within the method.

deleteState()

Deletes entries from the *CxCollabState* database table.

Syntax

```
public void deleteState(java.lang.String Verb,
                       java.lang.String BusinessObjectType,
                       java.lang.String PropDocID,
                       int stateValue)
```

Parameters

Verb Specifies the verb to use. In this case, the value must be *Delete*.

BusinessObjectType

Specifies the type of the business object.

PropDocID

Specifies identifier for the business object.

stateValue

Specifies the numeric tag for the state.

Exceptions

Throws the following exceptions:

- *CwDBSQLException*—Thrown if the *StateManagement* class is unable to execute the SQL query.
- *CollaborationException*—Thrown if the values of the parameters passed are null, or of any other unknown exception occurs within the method.

persistBO()

Persists a business object in the *CxCollabStateBO* database table.

Syntax

```
public void persistBO(java.lang.String CollabObjName,
                     java.lang.String Verb,
                     java.lang.String PropDocID,
                     java.lang.String BusinessObjectType,
                     com.crossworlds.BusObj BOtoSave)
```

Parameters

CollabObjName

Specifies the name of the collaboration.

Verb Specifies the verb to use. In this case, the value must be Create.

BusinessObjectType
Specifies the type of the business object.

PropDocID
Specifies identifier for the business object.

BOtoSave
Specifies the name of the business object to be persisted.

Exceptions

Throws the following exceptions:

- `CwDBSQLException`—Thrown if the `StateManagement` class is unable to execute the SQL query.
- `CollaborationException`—Thrown if the values of the parameters passed are null, or of any other unknown exception occurs within the method.
- `BOFormatException`—Thrown if there is an error in the data handler during the conversion of the business object to a recoverable string.

Notes

The `persistBO()` method can be used with internationalized collaborations. Both DBCS and MBCS characters can be stored correctly in the `CxCollabStateBO` database table. In addition, a `Locale` column has been added to the `CxCollabStateBO` table to store the locale information.

recoverBO()

Recovers a business object that was persisted in the `CxCollabStateBO` database table.

Syntax

```
public com.crossworlds.BusObj recoverBO(java.lang.String Verb,  
                                       java.lang.String PropDocID,  
                                       java.lang.String BusinessObjectType)
```

Parameters

Verb Specifies the verb to use. In this case, the value must be Retrieve.

BusinessObjectType
Specifies the type of the business object.

PropDocID
Specifies identifier for the business object.

Return values

Returns the stored business object from the `CxCollabStateBO` database table.

Exceptions

Throws the following exceptions:

- `CwDBSQLException`—Thrown if the `StateManagement` class is unable to execute the SQL query.
- `CollaborationException`—Thrown if the values of the parameters passed are null, or of any other unknown exception occurs within the method.

- `BOFormatException`—Thrown if there is an error in the data handler while converting the recovered string to an InterChange Server business object.

releaseDBConnection()

Releases the database connection.

Syntax

```
public void releaseDBConnection()
```

Notes

A connection acquired through the `getDBConnection()` method cannot be saved in a static variable for later reuse. A connection is always implicitly released at the end of the process to avoid connection leaks. In order for this to happen, the connection object used inside the `StateManagement` class must be released first with the `releaseDBConnection()` method.

resetData()

Resets the value of the boolean variable `bTranStarted`.

Syntax

```
public void resetData()
```

retrieveState()

Retrieves the latest value of the retry count stored in the `CxCollabState` database table.

Syntax

```
public int retrieveState(java.lang.String Verb,  
                        java.lang.String BusinessObjectType,  
                        java.lang.String PropDocID,  
                        int State)
```

Parameters

Verb Specifies the verb to use. In this case, the value must be `Retrieve`.

BusinessObjectType
Specifies the type of the business object.

PropDocID
Specifies identifier for the business object.

State Specifies the state value.

Exceptions

Throws the following exceptions:

- `CwDBSQLException`—Thrown if the `StateManagement` class is unable to execute the stored procedure.
- `CollaborationException`—Thrown if the values of the parameters passed are null, or of any other unknown exception occurs within the method.

saveState()

Saves the collaboration process parameters in the CxCollabState database table.

Syntax

```
public void saveState(java.lang.String CollabObjName,
                    java.lang.String Verb,
                    java.lang.String BusinessObjectType,
                    java.lang.String PropDocID,
                    int stateValue,
                    int retry,
                    double hours_to_timeout)
```

Parameters

CollabObjName

Specifies the name of the collaboration.

Verb

Specifies the verb to use. In this case, the value must be Create.

BusinessObjectType

Specifies the type of the business object.

PropDocID

Specifies the identifier for the business object.

stateValue

Specifies the numeric tag for the state.

retry

Specifies the retry value for the business object.

hours_to_timeout

Specifies the number of hours before the process times out; the data is stored in the CxCollabState table as a date data type.

Exceptions

Throws the following exceptions:

- *CwDBSQLException*—Thrown if the *StateManagement* class is unable to execute the SQL query.
- *CollaborationException*—Thrown if the values of the parameters passed are null, or of any other unknown exception occurs within the method.

setDBConnection()

Sets a database connection.

Syntax

```
public void setDBConnection(com.crossworlds.CwDBConnection DBConn)
```

Parameters

DBConn

Specifies the database connection.

Exceptions

Throws the *CwDBConnectionFactoryException* if the value of *DBConn* is null.

StateManagement()

Creates a StateManagement object and initializes it.

Syntax

```
StateManagement()
```

updateBO()

Updates a persisted business object in the CxCollabStateBO database table. The business object is passed through the delimited data handler and converted into a string. The resulting string is then stored in the database table.

Syntax

```
public void updateBO(java.lang.String CollabObjName,  
                    java.lang.String Verb,  
                    java.lang.String PropDocID,  
                    java.lang.String BusinessObjectType,  
                    com.crossworlds.BusObj BtoUpdate)
```

Parameters

CollabObjName

Specifies the name of the collaboration.

Verb

Specifies the verb to use. In this case, the value must be Update.

BusinessObjectType

Specifies the type of the business object.

PropDocID

Specifies identifier for the business object.

BtoUpdate

Specifies the name of the business object to be updated.

Exceptions

Throws the following exceptions:

- CwDBSQLException—Thrown if the StateManagement class is unable to execute the SQL query.
- CollaborationException—Thrown if the values of the parameters passed are null, or of any other unknown exception occurs within the method.
- BOFormatException—Thrown if there is an error in the data handler during the conversion of the business object to a string.

Notes

The updateBO() method can be used with internationalized collaborations. The CxCollabStateBO database table correctly stores DBCS and MBCS characters, and it contains a Locale column to store locale information.

updateState()

Updates the retry count value in the CxCollabState database table. When a two-hour timeout occurs and a retry is performed, this method updates the retry count in the 24-hour timeout row of the table.

Syntax

```
public void updateState(java.lang.String CollabObjName,
                        java.lang.String Verb,
                        java.lang.String BusinessObjectType,
                        java.lang.String PropDocID,
                        int stateValue,
                        int retry)
```

Parameters

CollabObjName

Specifies the name of the collaboration.

Verb

Specifies the verb to use. In this case, the value must be Update.

BusinessObjectType

Specifies the type of the business object.

PropDocID

Specifies the identifier for the business object.

stateValue

Specifies the numeric tag for the state.

retry

Specifies the retry value for the business object.

Exceptions

Throws the following exceptions:

- `CwDBSQLException`—Thrown if the `StateManagement` class is unable to execute the SQL query.
- `CollaborationException`—Thrown if the values of the parameters passed are null, or of any other unknown exception occurs within the method.

Part 5. Appendixes

Appendix. Standard Collaboration information

Because most CrossWorlds[®]-developed collaborations inherit business processes from the CollaborationFoundation template, this appendix explains the business processes that are common to all collaborations. Collaboration user documents refer to this document for an explanation of these processes. For information about CollaborationFoundation, see “CollaborationFoundation template” on page 29.

This appendix describes design guidelines whose use can help you achieve reusable, well-behaved collaborations. It covers the following topics:

- “Standard processes for collaboration templates”
- “Standard properties for collaboration templates” on page 468

Standard processes for collaboration templates

This section describes the business process flow of a typical WebSphere InterChange Server collaboration. The standard processes include the following:

- Retrieve process
- USE_RETRIEVE process
- Filtering process
- ADDITIONAL_RETRIEVE process
- Email process for error handling

The business process flow differs depending on the triggering verb and values of the collaboration object’s configuration properties.

Retrieve process

A collaboration can receive a triggering business object that uses the Retrieve verb. It is assumed that a source application’s business object uses this verb when its attributes contain cross-referenced key values. The business object’s map and the destination application’s connector use the primary key to return the business object from the destination application.

Figure 91 on page 462 illustrates a collaboration’s process for retrieving business objects when the triggering verb is Retrieve.

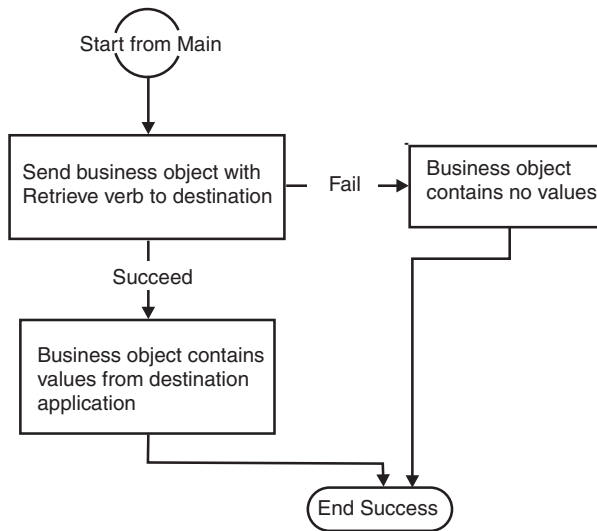


Figure 91. Retrieve process

Use_Retrieve process

Figure 92 on page 463 illustrates a collaboration's Use_Retrieve Process when its USE_RETRIEVE configuration property evaluates to true. For information about the configuration property, see its entry in TABLEA3.

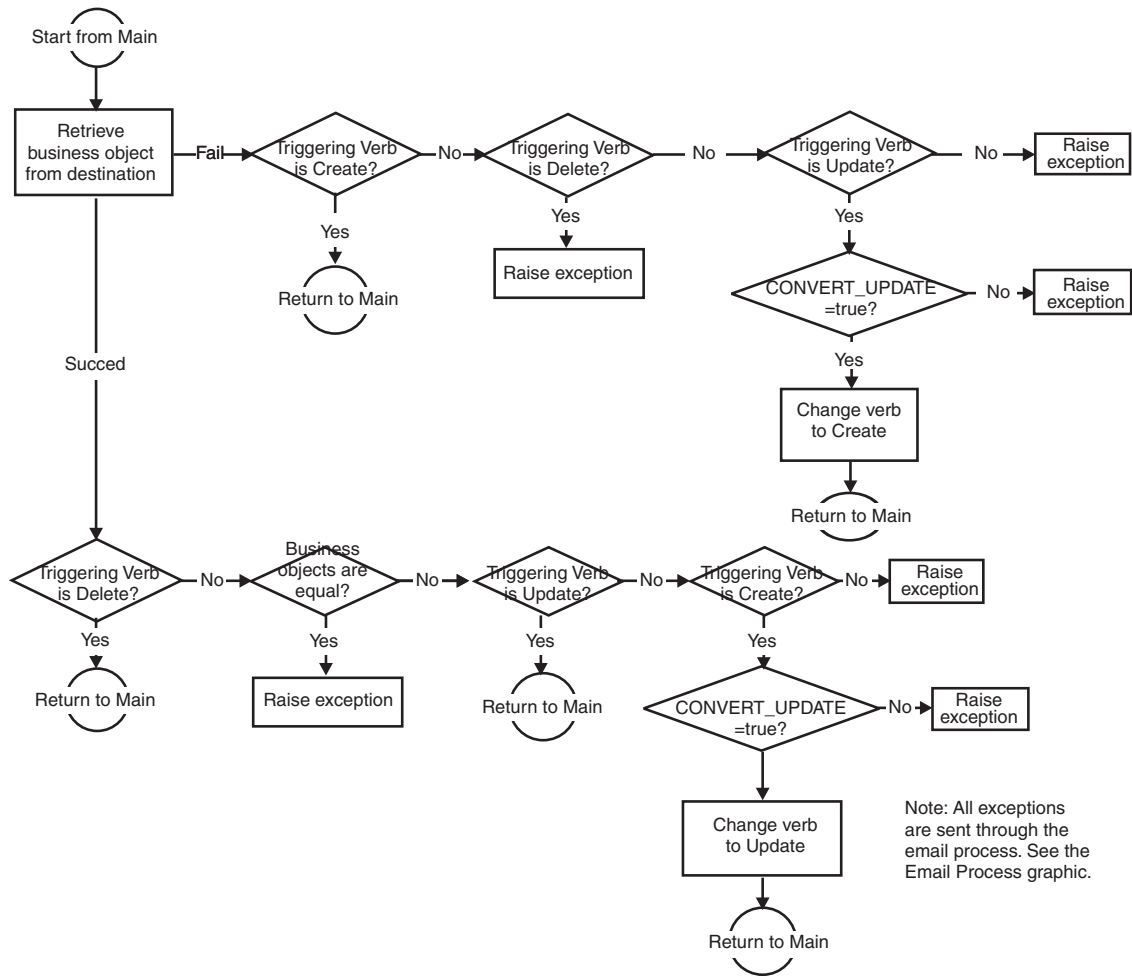


Figure 92. Use_Retrieve process

Table 102 on page 464 illustrates the relationship between the setting for the USE_RETRIEVE and CONVERT_CREATE properties when the triggering verb is Create. The relationship is similar between the USE_RETRIEVE and CONVERT_UPDATE properties.

Table 102. Relationship between the USE_RETRIEVE and CONVERT_CREATE properties

Value of USE_RETRIEVE	Value of CONVERT_CREATE	Action
true	True	<p>Retrieve the business object from the destination application.</p> <p>If the destination application does not return a business object, send the business object with the Create verb to the destination application.</p> <p>If the destination application returns a business object, check whether the source and destination business objects are identical:</p> <ul style="list-style-type: none"> • If they are identical, raise an exception stating that the business objects are identical and stop processing. • If they are not identical, convert the Create verb to the Update verb and send the business object to the destination.
false	True	<p>Send the business object with the create verb to the destination application.</p> <ul style="list-style-type: none"> • If the destination application fails to create the business object, convert the Create verb to the Update verb and resend the business object. • If the destination application succeeds in creating the business object, continue processing.
false	False	<p>Send the business object with the Create verb to the destination application.</p> <ul style="list-style-type: none"> • If the destination application fails to create the business object, raise an exception and stop processing. • If the destination application succeeds in creating the business object, continue processing.

Table 102. Relationship between the USE_RETRIEVE and CONVERT_CREATE properties (continued)

Value of USE_RETRIEVE	Value of CONVERT_CREATE	Action
true	False	<p>Retrieve the business object from the destination application.</p> <p>If the destination application does not return a business object, send the business object with the Create verb to the destination application.</p> <p>If the destination application returns a business object, determine whether the source and destination business objects are identical.</p> <ul style="list-style-type: none"> • If they are identical, raise an exception stating that the CONVERT_CREATE property value evaluates to false and stop processing. • If they are not identical, raise an exception and stop processing.

Filtering process

Figure 93 on page 466 illustrates a collaboration's process for filtering business objects based on values in specified attributes.

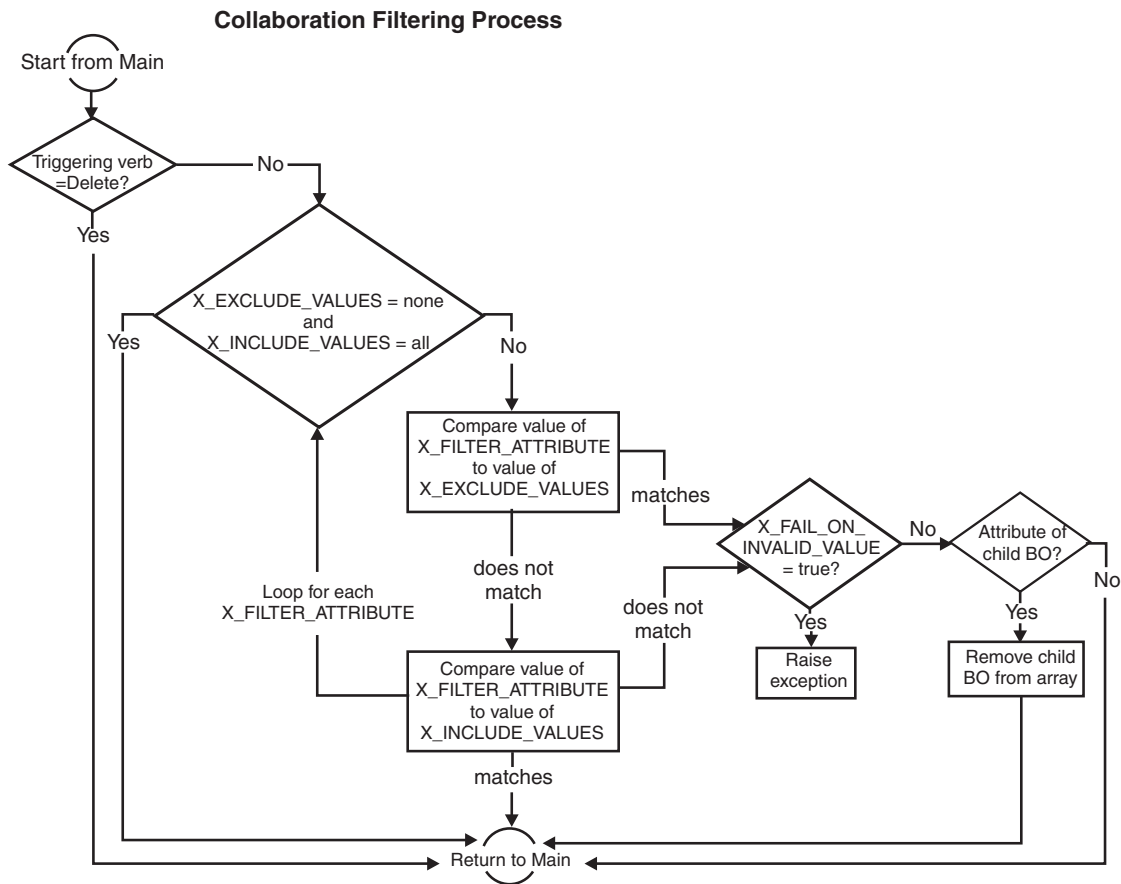


Figure 93. Filtering process

The above graphic refers to filter configuration properties with an “X_” rather than “1_” because you can add sets of filtering configuration properties to the collaboration in order to filter on multiple attributes. For more information on adding filter properties, see “Filtering data in the triggering business object” on page 32 and Table 103 on page 468.

To understand the collaboration’s behavior when 1_FAIL_ON_INVALID_VALUE evaluates to false, assume that a top-level business object named BusObjA contains an array of child business objects named BusObjB. Assume further that each BusObjB child business object contains an array of child business objects named BusObjC. To filter on an attribute named Type in the BusObjB business object, set the value of 1_FILTER_ATTRIBUTE to BusObjA.BusObjB.Type. To exclude all business objects whose Type contains the value of “Non-standard”, set the value of the 1_EXCLUDE_VALUES property to “Non-standard”.

Now assume that one of the business objects in the array of BusObjB has a value of “Non-standard” in its Type attribute and that 1_FAIL_ON_INVALID_VALUE evaluates to false. The collaboration removes this business object and its child business objects from processing.

Note: If the attribute that fails filtering is contained by the top-level business object, the collaboration continues processing the entire business object.

Additional_Retrieve process

If a collaboration's `ADDITIONAL_RETRIEVE` property evaluates to true, the collaboration retrieves the business object from the destination application after it has successfully synchronized the data. This property is useful when the source application requires a full-valued business object to be returned from the destination application, but the destination application's connector does not send back a complete business object after creating or updating its data. For more information on the `ADDITIONAL_RETRIEVE` configuration property, see "Standard properties for collaboration templates" on page 468.

Figure 94 illustrates a collaboration's Additional Retrieve process.

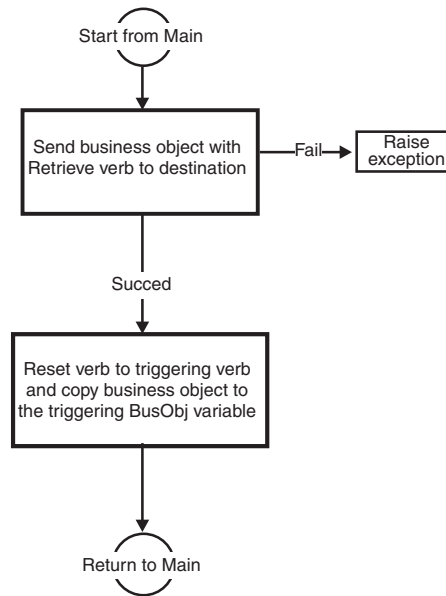


Figure 94. Additional Retrieve process

Email process for error handling

The settings of a collaboration's `SEND_EMAIL` and `INFORMATIONAL_EXCEPTIONS` properties determine whether the collaboration does one of the following:

- Sends email to a specified address when errors occur
- Ends in failure when a particular exception occurs

Figure 95 on page 468 illustrates a collaboration's process for sending email and determining whether a particular exception can cause the collaboration to end in failure.

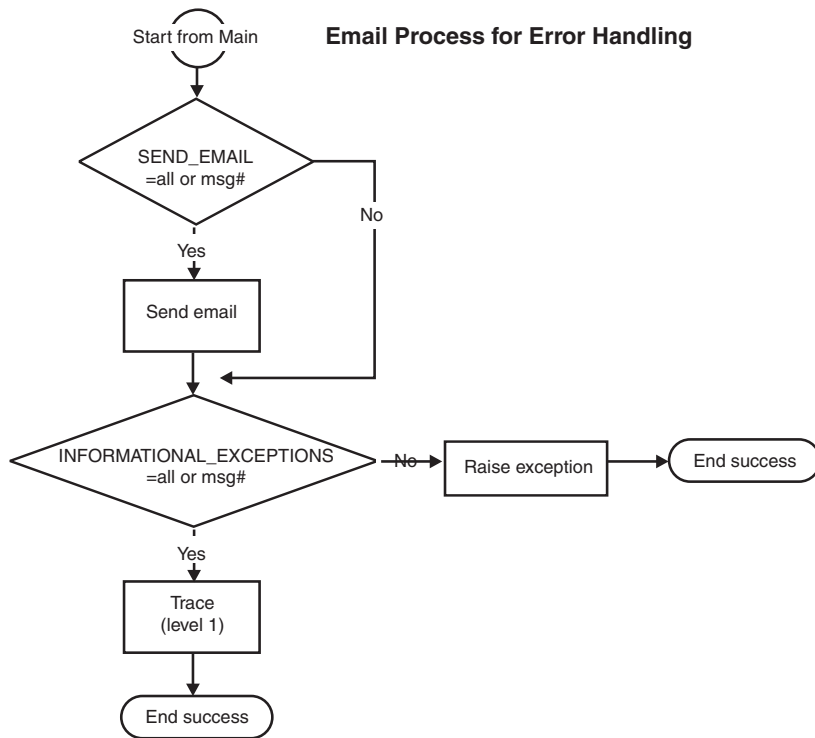


Figure 95. Email process for error handling

Standard properties for collaboration templates

Table 103 lists the standard configuration properties for collaboration templates. These properties are based on the CollaborationFoundation template.

Table 103. Standard properties for collaboration templates

Standard configuration property	Description
1_EXCLUDE_VALUES	Specifies values for the collaboration to use as a filter to prevent synchronization of the triggering business object.
1_FAIL_ON_INVALID_VALUE	Indicates how the collaboration handles the value of an attribute specified in 1_FILTER_ATTRIBUTE that does not meet the filtering criteria.
1_FILTER_ATTRIBUTE	Specifies the business object attribute for the collaboration to use when comparing values specified in the 1_EXCLUDE_VALUES and 1_INCLUDE_VALUES properties.
1_INCLUDE_VALUES	Specifies values for the collaboration to use as a filter to allow synchronization of the triggering business object.
ADDITIONAL_RETRIEVE	Indicates whether the collaboration retrieves the business object from the destination application after it successfully synchronizes its data.

Table 103. Standard properties for collaboration templates (continued)

Standard configuration property	Description
CONVERT_CREATE	Indicates how the collaboration responds to a Create request if the triggering business object that has been created in the source application already exists in the destination application.
CONVERT_UPDATE	Indicates how the collaboration responds to an Update request if the triggering business object that has been updated in the source application does not exist in the destination application.
INFORMATIONAL_EXCEPTIONS	Specifies how the collaboration responds to an exception.
SEND_EMAIL	Indicates whether the collaboration sends email when it receives an exception.
USE_RETRIEVE	Indicates whether the collaboration retrieves its triggering business object from the destination application before it synchronizes data.

1_EXCLUDE_VALUES

The 1_EXCLUDE_VALUES collaboration configuration property specifies values for the collaboration to use as a filter to prevent synchronization of the triggering business object. The collaboration excludes a business object from synchronization if the value of the attribute specified by 1_FILTER_ATTRIBUTE matches one of the values listed in this property.

The possible values for 1_EXCLUDE_VALUES are as follows:

- A comma-delimited list of values to exclude (such as "in-stock, at-customer").
- The value none, which indicates that no objects are to be excluded from processing.
- No value, which is useful to exclude an attribute whose value is either a null (CxIgnore) or an empty string (CxBlank).

The default value is none.

You can delete this property from the list of configuration properties available for the collaboration template. When the 1_EXCLUDE_VALUES property is deleted, it indicates that the property is not used in the filtering process.

Note that the collaboration is not sensitive to the case of the specified values, nor is it sensitive to extra spaces in a list of values.

Specify values for this property if the list of exclusion values is *shorter* than the list of inclusion values. Otherwise, specify values for 1_INCLUDE_VALUES. If specifying values in 1_INCLUDE_VALUES, set the value of 1_EXCLUDE_VALUES to none.

1_FAIL_ON_INVALID_VALUE

The 1_FAIL_ON_INVALID_VALUE collaboration configuration property indicates how the collaboration handles the value of an attribute specified in

1_FILTER_ATTRIBUTE that does not meet the filtering criteria. Set the 1_FAIL_ON_INVALID_VALUE property to one of the following values:

- true—Set to true to cause the collaboration to raise an exception and stop processing if the value of the attribute specified in 1_FILTER_ATTRIBUTE does not meet the filtering criteria specified in 1_EXCLUDE_VALUES or 1_INCLUDE_VALUES.
- false—Set to false to cause the collaboration to perform one of the following processes when the specified attribute's value does not pass the filtering criteria:
 - If the attribute specified in 1_FILTER_ATTRIBUTE is part of the parent business object, the collaboration continues processing.
 - If the attribute specified in 1_FILTER_ATTRIBUTE is part a child business object, the collaboration removes the child business object that contains the attribute as well as all of its children; the collaboration continues processing.

The default value is true. The values are not case sensitive.

You can delete this property from the list of configuration properties available to the collaboration template. Deleting the property indicates that it is not used in the filtering process.

1_FILTER_ATTRIBUTE

The 1_FILTER_ATTRIBUTE collaboration configuration property specifies the business object attribute for the collaboration to use when comparing values specified in the 1_EXCLUDE_VALUES and 1_INCLUDE_VALUES properties. The collaboration compares the value in the specified attribute against the values specified for exclusion or inclusion to prevent or ensure synchronization of business objects with specific values. For example, assume that a top-level business object named BusObjA contains a child business object named BusObjB. To filter on an attribute named Type in the BusObjB business object, set the value of 1_FILTER_ATTRIBUTE to BusObjA.BusObjB.Type.

Note: The collaboration evaluates this property only if its 1_EXCLUDE_VALUES evaluates to a value other than none or its 1_INCLUDE_VALUES evaluates to a value other than all.

Valid values for this collaboration configuration property are as follows:

- The exact name of the business object and its attribute as it is defined in WebSphere InterChange Server (for example, InstallShipment.ShipmentType). To determine the attribute's name, do one of the following:
 - View the business object in System Manager.
 - View the repository file for the business object, which is named repository\BO_*BusinessObjectName*.txt

Note: In this document backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute backslashes with slashes (/). All file pathnames are relative to the directory where the product is installed on your system.

- *BusinessObjectName*.ObjectEventId, which is the attribute that contains a unique, internal identifier for each business object. This is the default value for 1_FILTER_ATTRIBUTE.

The collaboration template is designed to filter on any number of business object attributes. To add more filters, add a set of configuration properties (X_FILTER_ATTRIBUTE, X_FAIL_ON_INVALID_VALUE, and either

X_EXCLUDE_VALUES or X_INCLUDE_VALUES) to the collaboration, where X is an integer from 2 to N. Additional properties must be added in order from 2 to N and must have the exact names listed above.

1_INCLUDE_VALUES

The 1_INCLUDE_VALUES collaboration configuration property specifies values for the collaboration to use as a filter to allow synchronization of the triggering business object. The collaboration synchronizes only business objects with the specified values if the value of the attribute specified by 1_FILTER_ATTRIBUTE matches one of the values listed in this property.

Valid values for the 1_INCLUDE_VALUES property are as follows:

- A comma-delimited list of values to include (such as “standard, in-stock, at-customer”).
- The value all, which indicates that all objects should be included in processing.

The default value is all. Note that the values do not have to be case sensitive. In addition, the collaboration ignores extra spaces in a list of values.

If this property is not to be used in the filtering process, you can delete it from the list of configuration properties available to the collaboration.

Specify values for this property if the list of inclusion values is *shorter* than the list of exclusion values. Otherwise, specify values for 1_EXCLUDE_VALUES. If specifying values in 1_EXCLUDE_VALUES, set the value of 1_INCLUDE_VALUES to all.

ADDITIONAL_RETRIEVE

The ADDITIONAL_RETRIEVE collaboration configuration property indicates whether the collaboration retrieves the business object from the destination application after it successfully synchronizes its data. Set the ADDITIONAL_RETRIEVE property as follows:

- Set to true to cause the collaboration to retrieve the business object from the destination application after it has successfully synchronized its data. This setting is useful when the source application requires a full-valued business object to be returned from the destination application, but the destination application’s connector does not return a complete business object after creating or updating its data.
 - If the collaboration successfully retrieves the full-valued business object, it saves it in the triggeringBusObj variable to return it to the source application. If the collaboration is bound to a source connector that makes a synchronous request, the triggering business object’s values are returned to the source connector as soon as the collaboration completes processing. The values are returned as if the business object’s values had been passed by reference.

Note: The collaboration also resets the verb of the triggeringBusObj variable from Retrieve to the original verb. Resetting the verb allows the collaboration to use the original triggering verb if required in the Additional Processing 5 subprocess. For more information about the Additional Retrieve process, see “Additional_Retrieve process” on page 467.

- If the collaboration does not successfully retrieve the full-valued business object, it raises an exception. Exception handling is determined by the setting of the INFORMATIONAL_EXCEPTIONS property.

- Set to false to cause the collaboration not to retrieve the business object from the destination application after it has successfully synchronized its data.

The default value is false.

CONVERT_CREATE

The CONVERT_CREATE collaboration configuration property indicates how the collaboration responds to a Create request if the triggering business object that has been created in the source application already exists in the destination application. Set the CONVERT_CREATE property as follows:

- Set to true to cause the collaboration to convert a Create request to an Update request if the triggering business object that has been created in the source application already exists in the destination.
- Set to false to cause the collaboration to raise an exception. Exception handling is determined by the setting of the INFORMATIONAL_EXCEPTIONS property.

The default value is false.

The actions that the collaboration takes for CONVERT_CREATE depend on the value of the USE_RETRIEVE collaboration configuration property.

- If USE_RETRIEVE evaluates to true, the collaboration evaluates the CONVERT_CREATE property only if the triggering verb is Create and the collaboration succeeded in retrieving the business object.
- If USE_RETRIEVE evaluates to false, the collaboration evaluates the CONVERT_CREATE property only after it fails in its attempt to create the triggering business object in the destination.

For a table that illustrates the relationship between the USE_RETRIEVE and CONVERT_CREATE configuration properties, see “Use_Retrieve process” on page 462.

CONVERT_UPDATE

The CONVERT_UPDATE collaboration configuration property indicates how the collaboration responds to an Update request if the triggering business object that has been updated in the source application does not exist in the destination application. Set the CONVERT_UPDATE property as follows:

- Set to true to cause the collaboration to convert an Update request to a Create request if the triggering business object that has been updated in the source application does not exist in the destination.
- Set to false to cause the collaboration to raise an exception. Exception handling is determined by the setting of the INFORMATIONAL_EXCEPTIONS property.

The default value is false.

The actions that the collaboration takes for CONVERT_UPDATE depend on the value of the USE_RETRIEVE collaboration configuration property:

- If USE_RETRIEVE evaluates to true, the collaboration evaluates the CONVERT_UPDATE property only if the triggering verb is Update and the retrieve ascertains that the business object does not exist in the destination.
- If USE_RETRIEVE evaluates to false, the collaboration evaluates the CONVERT_UPDATE property only after it fails in its attempt to update the triggering business object in the destination.

INFORMATIONAL_EXCEPTIONS

The INFORMATIONAL_EXCEPTIONS collaboration configuration property indicates how the collaboration responds to an exception. The property accepts the following values:

- All—Whenever an exception occurs, the collaboration sends it to a trace and ends in success.
- None—Whenever an exception occurs, the collaboration raises it and ends in failure.
- A comma-delimited list of message numbers—Whenever one of the listed exceptions occurs, the collaboration sends the exception to a trace and ends in success. The exception message numbers correspond to those in the collaboration's message file (collaborations\messages*CollaborationName.txt*).

The default value is 1000, 2000, 2005, 2010, 2015, 2020, 3000, 3010, 3020.

SEND_EMAIL

The SEND_EMAIL collaboration configuration property indicates whether the collaboration sends email when it receives an exception. Set the SEND_EMAIL property as follows:

- All—Whenever an exception occurs, the collaboration sends email.
- None—Whenever an exception occurs, the collaboration does not send email.
- A comma-delimited list of message numbers—Whenever one of the listed exceptions occurs, the collaboration sends email. The exception message numbers correspond to those in the collaboration's message file (collaborations\messages*CollaborationName.txt*).
- A range of message numbers—Whenever one of the exceptions located in the specified range occurs, the collaboration sends email.

The default value is none.

Note: Specify the address in the **Email notification address** field on the Collaboration Object Properties dialog box in System Manager. If an email address is specified for a collaboration, the collaboration calls the Email collaboration when an error occurs. The Email collaboration can be installed and configured as part of WebSphere InterChange Server.

USE_RETRIEVE

The USE_RETRIEVE collaboration configuration property indicates whether the collaboration retrieves its triggering business object from the destination application before it synchronizes data. Set the USE_RETRIEVE property as follows:

- Set to true to cause the collaboration to retrieve its triggering business object from the destination application before synchronizing data. This setting is useful when:
 - The collaboration is triggered by another collaboration and might be required to perform compensation processing if any step of the grouped collaborations fails. Forcing the collaboration to retrieve values before changing them allows the collaboration to restore the original values in an Update or Delete process.
 - The collaboration is triggered by a Wrapper Collaboration, which always sends the triggering business object with the Create verb. Setting this property to true prevents the collaboration from failing if the business object already exists in the destination. When USE_RETRIEVE evaluates to true, the

collaboration compares values in the source and destination applications before performing a Create or Update. If the objects are identical, it ends successfully.

- Set to false to cause the collaboration to begin processing without retrieving its triggering business object from the destination application. This setting is useful when it is not important to compare the values in the source and destination business objects.

The default value is false.

Note: For more information on using this property, see “`CONVERT_CREATE`” on page 472 and “`CONVERT_UPDATE`” on page 472.

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
CICS
CrossWorlds
DB2
DB2 Universal Database
Domino
IMS
Informix
iSeries
Lotus
Lotus Notes
MQIntegrator
MQSeries
MVS
OS/400
Passport Advantage
SupportPac
WebSphere
z/OS

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.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

WebSphere InterChange Server includes software developed by the Eclipse Project (<http://www.eclipse.org/>).

IBM WebSphere InterChange Server v4.3.0, IBM WebSphere Business Integration Toolset v4.3.0



Glossary

action. An activity diagram symbol that indicates a single step in the business process. An action node contains a code fragment.

activity diagram. Defines the control flow for a collaboration's scenario. An activity diagram consists of a set of symbols that specify the actions required by the business process and the logic that determines the order in which the actions execute.

attribute. A data item in a business object.

BaseCollaboration. A WebSphere InterChange Server-defined class from which all other collaboration classes are derived. The BaseCollaboration class contains methods for manipulating a collaboration.

binding . The act of attaching a collaboration object to the objects that can supply business objects to it or receive business objects from it. The objects to which a collaboration attaches can be connectors or other collaboration objects.

break. A symbol placed in an iterator's activity diagram to force premature termination of the iteration.

business object. A set of data that represents a business entity, along with a verb that indicates an action on the data.

business object definition. A description of the format and data contained in a business object. A business object definition contains a name, a version, a set of supported verbs, and an ordered set of attributes.

business object probe. Monitors and reports the values of a specified business object's attributes during runtime. Business object probes can be placed on any transition link (with the exception of a decision node's incoming transition link and a service call link).

BusObj. A WebSphere InterChange Server-defined class that represents a business object.

BusObjArray. A WebSphere InterChange Server-defined class that represents an array of business objects. BusObjArray is used for a business object attribute whose value is a reference to an array of child business objects.

code fragment. The specification of an action via a series of code statement, using the collaboration API or other Java code.

collaboration. Business logic that describes a generic distributed business process. A collaboration interacts

with individual applications, tying together the events and data of these different applications and extending their functionality.

collaboration group. An executable set of collaboration objects, formed by binding collaboration objects.

collaboration object. An object created from a collaboration template. A collaboration object is executable when it is configured and bound to applications, represented by connectors, or to other collaborations.

collaboration configuration property. Any configurable information about a WebSphere InterChange Server object. A collaboration template has standard properties and collaboration-specific properties. A collaboration developer creates collaboration-specific properties to enable an administrator to specify some aspect of the collaboration object's runtime behavior.

collaboration template. The logic and framework of a collaboration. A collaboration template provides the definition of a collaboration and from which a collaboration object may be instantiated; a collaboration template itself is never executable.

CollaborationException. A WebSphere InterChange Server-defined exception object.

compensation. The action that a collaboration takes during rollback of a transaction to undo a previously-executed service call.

control flow. The flow of business process logic. Within collaborations, an activity diagram defines the control flow for a particular scenario, specifying the actions required for the business process. Decision nodes and iterators are used within the activity diagram to further specify the execution order of the action nodes.

correlation attribute. Identifies a conversation between two business processes when a collaboration is used as a long-lived business process. Correlation attributes are initialized by a start node or outbound service call; they can then be used by conversation participants to make external calls or to receive a matching event from external sources.

currentException. A WebSphere InterChange Server-defined variable that holds the value of the preceding exception. The scope of currentException is raised in the preceding action, subactivity, or iterator.

decision node. A node that handles decision branching in a scenario. Decision nodes are used when there are multiple possible outcomes of an action, subdiagram, or iterator node. Each branch in a decision node has a condition, and the control flow shifts to the branch whose condition evaluates to true.

declaration. The name and type of a variable that you intend to use. The compiler requires a declaration for each variable used.

event isolation. The assurance that multiple collaborations do not concurrently process events that relate to the same business object data.

event sequencing. InterChange Server's assurance that a collaboration processes multiple events that relate to the same business object one at a time, in the same order in which the events arrive.

exception. An object used to pass a runtime error to some other entity that can handle the error. In an activity diagram, an exception is caught on an exception transition link.

failed event. This term has been changed. See unresolved flow.

full-valued business object. A business object that has data values for more attributes than just primary key attributes.

import statement. A Java statement that includes a class or a package of classes into the collaboration class.

iterator. An activity diagram symbol that embeds a reference to a nested diagram that implements a looping operation, and the diagram that contains the looping behavior. An iterator can loop through all attributes in a business object or through all elements of a business object array.

key values. The values of attributes that typically comprise the unique identification of a business object or the associated application entity.

long-lived business process. A method of configuring and deploying a collaboration to enable asynchronous communication between business processes. In a long-lived business process, event flow context persists throughout the duration of a service call.

minimum transaction level. The transaction level set by a collaboration template developer, indicating the level of transaction services required for executing collaboration objects created from the template.

package. A group of related Java classes. A collaboration template can be made part of a package and it can import other packages.

port. The interface between a collaboration and other objects in the WebSphere InterChange Server system. It

is through a port that a collaboration object binds with a connector or with another collaboration object.

reference-valued business object. A business object that contains values for only its key attributes. It does not contain values for non-key attributes.

scenario. The code that handles one or more incoming events. Scenarios can be used to partition a collaboration's logic.

scenario tree. The set of scenarios, displayed hierarchically, that includes composite scenarios, subdiagrams, and iterators.

scenario variable. A variable whose scope extends to all parts of all diagrams in a scenario.

service call. An activity diagram symbol that represents a request to a WebSphere InterChange Server object outside the collaboration, such as a connector or another collaboration.

subdiagram. An activity diagram symbol that represents another, nested, activity diagram, and the nested diagram itself.

template variable. A variable whose scope extends to all scenarios in a collaboration template.

template tree view . The tree viewer that displays the template definitions, scenario tree, and message file of the collaboration template. Display of the template tree view is optional.

transactional collaboration. A collaboration that follows the database transactional model and provides for data consistency for business processes. A transactional collaboration is capable of rolling back when a runtime error causes the collaboration object to fail. In a transactional collaboration, the service calls have compensation defined.

transition link. An activity diagram symbol that indicates control flow between other symbols of an activity diagram.

transactional verb. A business object verb that indicates a data modification, such as Create, Update, or Delete. Retrieve is not a transactional verb because it does not modify data.

triggering event. The business object that a connector sends to a subscribing collaboration when an application event occurs.

triggeringBusObj. The Designer-declared variable that contains a scenario's triggering event when the scenario starts to execute.

UID. A unique identifier for each symbol in the activity diagrams of a scenario.

unresolved flow. The business object whose receipt caused a collaboration to execute a scenario that ended unsuccessfully. An unresolved flow can be a failed flow (a flow that failed due to application or logic problems), a deferred flow (a flow whose recovery has been deferred), an in-transit flow (a flow created when the server crashes during a service call transmission in a collaboration configured for Service Call In-Transit persistence), or a possible duplicate flow (a flow that may have already been received by the collaboration).

Index

Special characters

- .class file 6, 82, 86
- .java file 6, 82, 86
- .txt file 7
- (Base Collaboration), `getCurrentLoopIndex()` method 347
- (Base Collaboration), `getLocale()` method 349
- (BaseCollaboration), `getLocale()` method 67
- (BaseCollaboration), `getMessage()` method 66, 350
- (BusObj), `getLocale()` method 68
- (symbol), UID 112

Numerics

- 1_EXCLUDE_VALUES property 469
- 1_FAIL_ON_INVALID_VALUE property 469
- 1_FILTER_ATTRIBUTE property 470
- 1_INCLUDE_VALUES property 471

A

- access, data 30
- action 5
 - adding activity definitions to action nodes 114
 - adding to an activity diagram 113
 - collaboration properties and 142
 - defined 5, 112, 479
 - defining properties for action nodes 113
 - name of action node 113
 - restrictions on links 117
 - using a service call with 113
- Action Properties dialog 113
 - enabling direct editing of Code Fragment window 114
- activity definition
 - adding to an action node 114
 - defined 114
 - using web services in 116
- activity definitions
 - tags for 156
 - using web services 116
- Activity definitions 154
- activity diagram 4
 - adding a decision node 122
 - adding action to 113
 - adding and modifying transition links in 116
 - adding failed termination to 144
 - adding iterator to 139
 - adding service call to 127
 - adding subdiagram to 136
 - adding successful termination to 143
 - adding text to 142
 - arrangement of 135
 - cancelling operation in 142
 - closing 145
 - copying contents of 146
 - creating 183
 - defined 5, 100, 109, 479
 - deleting 146
 - development styles 112
 - documenting 145
 - enlarging 22
 - activity diagram (*continued*)
 - example of 5
 - exceptions and 167
 - finding text in 21
 - finding UID of 21
 - handling triggering event 205
 - locking 22
 - main 134, 135, 143, 169, 170
 - menus for 109
 - opening 22, 145
 - printing 22
 - read-only mode 22
 - refreshing display of 22
 - resizing 22
 - role of links 116
 - saving 145
 - saving as text 23
 - selecting all nodes in 21
 - start and end symbols in 110
 - stopping execution of 167
 - symbols in 110
- Activity Editor 114
 - Graphical view 149
 - Java view 151
 - keyboard shortcuts 153
 - menus 151
 - starting 149
- Add Day 311
- Add Element 261, 333
- Add Month 311
- Add Year 312
- `addElement()` method 386
- Additional_Retrieve process 467
- ADDITIONAL_RETRIEVE property 471
- aligning scenarios 190
- Alignment toolbar 21, 23
- AND operator 124
- AnyException 274
- AnyException exception 168, 173, 358, 426
- API
 - Collaboration 171
 - Exceptions from the Collaboration 181
- AppBusObjDoesNotExist exception subtype 425
- Append Text 325
- AppLogOnFailure exception subtype 424
- AppMultipleHits exception subtype 305, 424
- AppRequestNotYetSent exception subtype 181, 425
- AppRetrieveByContentFailed exception subtype 424
- AppTimeOut exception subtype 424
- AppUnknown exception subtype 179, 425
- arithmetic operator 116
- array
 - `EventKeyAttrDef()` 430
 - `EventQueryDef()` 431
 - `FailedEventInfo()` 433
- assignment operator 116
- asynchronous inbound service call 127
 - matching correlation attributes in 133
- asynchronous outbound service call 127
 - setting correlation attributes for 133

- attribute
 - checking for key 376
 - data type of 383
 - defined 479
 - required 378
- attribute value
 - adding together 396
 - basic data type 208, 210
 - blank 376
 - checking existence of 372
 - comparing 370
 - default 380
 - null 207, 212, 377
 - retrieving 208, 372
 - retrieving as string 382
 - retrieving maximum 388, 389, 390
 - retrieving minimum 391, 392, 393
 - setting 209, 379, 382
 - setting default value for 380
 - using 207
 - validating data type 383
 - zero-length string 376
- AttributeException 274
- AttributeException exception 168, 358, 426

B

- BaseCollaboration class 10, 343, 367
 - defined 343, 479
 - dropFailedEvent() 343
 - dynamicSend() 344
 - existsConfigProperty() 345
 - getConfigProperty() 346
 - getConfigPropertyArray() 346
 - getCurrentLoopIndex() 347
 - getDBConnection() 347
 - getLocale() 67, 349
 - getMessage() 66, 350
 - getName() 351
 - implicitDBTransactionBracketing() 351
 - isCallerInRole() 352
 - isTraceEnabled() 352
 - logError() 353
 - logInfo() 353
 - logWarning() 353
 - method summary 343, 415
 - queryFailedEvents() 355
 - rai
 - See* xception()
 - resubmitFailedEvent() 360
 - saveFailedEvent() 362
 - sendEmail() 362
 - trace() 363
- Begin Transaction 285
- beginTransaction() 452
- beginTransaction() method 227
- beginTransaction() method 399
- Best-effort transaction level 85
- BiDiBOTransformation() 419
- BiDiBusObjTransformation() 420
- Bidirectional language parameter input dialog 72
- bidirectional languages, Designing collaborations for 69
- bidirectional languages, Enabling connectors for 70
- BiDiStringTransformation() 421
- BiDiTransformation parameter in the Connector Configurator, The 71
- binding 7

- binding (*continued*)
 - defined 479
- blank attribute value 376
- block
 - function 154
 - organization of 157
 - New Constant function 156
 - supported function 157
 - web services function 155
- Boolean class
 - as stored-procedure parameter type 411
- boolean data type 90, 208, 210, 224, 373, 379, 383
 - as stored-procedure parameter type 411
- boot-time recovery transport-related exceptions, Handling 180
- BPEL
 - exporting BPEL files from a template 102
 - importing BPEL files to a template 102
- branch
 - default branch 120, 123
 - exception branch 120, 123
 - normal branch 120, 122
 - number of decision node branches 120
 - types of branches in decision nodes 120
- branching 38
 - using collaboration properties to branch 38
 - using decision nodes 120
- break symbol 479
- business object
 - adding to an array 386
 - business object definition for 375
 - check for attribute 372
 - child 204, 208, 210
 - class for 11, 367
 - comparing attribute values 207, 370, 371
 - comparing key attribute values 369
 - copying 206, 368
 - creating 203
 - defined 479
 - dependent data in 39
 - duplicating 207, 369
 - full-valued 60, 480
 - generic Item business objects 42
 - in flow trigger 9
 - in triggering event 98
 - InstalledProduct business object 45
 - Item business object 45
 - key attribute in 376
 - locale 67
 - monitoring values with a business object probe 119
 - moving values from one to another 206
 - null attribute in 207, 212, 377
 - number in a business object array 395
 - operations on 203
 - recursive processing of 44, 45
 - reference-valued 60
 - removing from business object array 394
 - required attribute in 378
 - retrieving attribute value 372, 382
 - retrieving from business object array 387
 - retrieving key attribute value 378
 - setting attribute values 209, 379, 382
 - setting key values 380
 - setting value of 395
 - swapping in an array 396
 - triggering 30
 - validating attribute data type 383

- Business object
 - reference-valued 480
- business object array
 - adding attribute values together 396
 - adding business object to 386
 - child 204, 205
 - class for 11, 385
 - comparing with another 387
 - creating 205
 - duplicating 386
 - iterator and 139
 - removing all elements from 394
 - removing element from 394
 - retrieving a business object from 387
 - retrieving contents of 387
 - retrieving last index of 388
 - retrieving maximum attribute value from 388, 389, 390
 - retrieving minimum attribute value from 391, 392, 393
 - retrieving size of 395
 - retrieving values as string 397
 - reversing position of elements in 396
 - setting element of 395
- business object array function block
 - Add Element 261
 - Duplicate 262
 - Equals 262
 - Get Element At 263
 - Get Elements 263
 - Get Last Index 263
 - Is Business Object Array 264
 - Max Attribute Value 264
 - Max Business Object Array 265
 - Max Business Objects 266
 - Min Attribute Value 267
 - Min Business Object Array 267
 - Min Business Objects 268
 - Remove All Elements 269
 - Remove Element 269
 - Remove Element At 270
 - Set Element At 270
 - Size 271
 - Sum 271
 - Swap 272
 - To String 272
- Business object array function block 261
- business object definition
 - defined 479
 - retrieving name of 375
- Business object function blocks 237
- business object probe
 - adding to a transition link 119
 - defined 119, 479
 - functions of 119
- busobj class
 - constructor 203
- BusObj class 11, 367, 384
 - copy() 368
 - defined 367, 479
 - deprecated methods 384
 - duplicate() 369
 - equalKeys() 369
 - equals() 370
 - equalsShallow() 371
 - exists() 372
 - get() 372
 - getBoolean() 372
 - getBusObj() 372

- BusObj class (*continued*)
 - getBusObjArray() 372
 - getCount() 384
 - getDouble() 372
 - getFloat() 372
 - getInt() 372
 - getKeys() 384
 - getLocale() 68, 374
 - getLong() 372
 - getLongText() 372
 - getString() 372
 - getType() 375
 - getValues() 384
 - getVerb() 375
 - isBlank() 376
 - isKey() 376
 - isNull() 377
 - isRequired() 378
 - keysToString() 378
 - method summary 367
 - retrieving value for 373
 - set() 379, 384
 - setDefaultAttrValues() 380
 - setKeys() 380
 - setLocale() 68, 381
 - setVerb() 381
 - setWithCreate() 382
 - toString() 382
 - validData() 383
- BusObjArray class 11, 385, 397
 - addElement() 386
 - defined 385, 479
 - duplicate() 386
 - elementAt() 387
 - equals() 387
 - getElements() 387
 - getLastIndex() 388
 - max() 388
 - maxBusObjArray() 389
 - maxBusObjs() 390
 - method summary 385
 - min() 391
 - minBusObjArray() 392
 - minBusObjs() 393
 - removeAllElements() 394
 - removeElement() 394
 - removeElementAt() 394
 - retrieving value for 373
 - setElementAt() 395
 - size() 395
 - sum() 396
 - swap() 396
 - toString() 397

C

- call
 - asynchronous inbound service 127
 - asynchronous outbound service 127
 - dynamic service 54
 - synchronous service 127
- call requests, Unsent service 181
- call statement 220, 221
- CALL statement 287, 289, 402, 403
- called collaboration 52, 53
- caller collaboration 52, 53
- calling the map 200

- callMap() 443
- Catch Collaboration Exception 303
- Catch Error 334
- Catch Error Type 334
- catching the exception 172
- changing diagram display 188
- changing display
 - user preferences 187
- changing general display 187
- changing the color of symbols and links 189
- character encoding 64
 - design principles 69
- class
 - CollaborationException 167
 - cwdbstoredprocedureparam 222
 - cxexecutioncontext 202
 - enumeration 214
 - EventKeyAttrDef() 430
 - EventQueryDef() 431
 - FailedEventInfo() 433
 - filter 435
 - globals 441
 - java.sql.types 224
 - object 208, 210
 - SmartCollabService 445
 - StateManagement 451
- class, BaseCollaboration 343, 367
- Classes in the EventManagement package 429
- classes to InterChange Server
 - importing third-party 165
- CLASSPATH environment variable 86, 89, 166
- code fragment 6
 - defined 114, 479
 - enabling direct editing of Code Fragment window 114
 - using Activity Editor's Graphical view to add code 149
 - using Activity Editor's Java view to add code 151
- Coding to save a failed event 76
- collaboration
 - base class for 343
 - called 52, 53
 - caller 52, 53
 - calling map 199
 - defined 3, 479
 - deploying 6
 - designing 29
 - development process of 12
 - exception handling in 37
 - execution states 168
 - handling successful subdiagram 137
 - handling unsuccessful subdiagram 138
 - initializing 200
 - internationalized 63
 - operations on 193
 - parallel execution of 55
 - performance considerations 134
 - recovery of 170, 180
 - testing 12
 - transactional 128, 131, 143
 - unresolved flow 170
 - used as a long-lived business process 8
 - wrapper 39
- Collaboration API 10, 171, 423
 - BaseCollaboration 343
 - BusObj 367
 - BusObjArray 385
 - CollaborationException 167, 423
 - CwDBCConnection 399
 - Collaboration API (*continued*)
 - CwDBStoredProcedureParam 411
 - CxExecution 415
 - exceptions 423
 - Exceptions from the 181
 - collaboration configuration property
 - enableinstancereuse 199
 - collaboration configuration property
 - case-sensitivity 197
 - checking existence of 345
 - collaborationinstancecachesize 199
 - controlling flow 38
 - creating 91, 92
 - defined 479
 - deleting 94
 - enableinstancereuse 199
 - FAIL_ON_CONTACT_ERROR 41
 - FIND_ALL_ITEM_ERRORS 41
 - for delegating dependent data checking 40
 - for long-lived business processes 94
 - ITEM_TYPE 43
 - naming 35
 - obtaining value 197
 - obtaining value of 142, 346
 - types of 91
 - use with internationalized collaborations 68
 - VERIFY_SYNC_ 40
 - collaboration development
 - platform for 10
 - tools for 10
 - collaboration group 52
 - creating 53
 - defined 52, 479
 - example of 52, 61
 - using with long-lived business processes 52
 - collaboration locale 67, 349
 - collaboration object 7
 - binding 7
 - class for 10, 343
 - configuring 7
 - defined 7, 479
 - in-doubt 170
 - name of 351
 - recycling 198
 - reusing 198
 - running in threads 8
 - transaction programming model 225, 351
 - collaboration property for the map name, creating a 200
 - collaboration runtime environment
 - handling exceptions 137, 138
 - Java exceptions 172
 - processing action 126
 - processing service call 126
 - tracing component 197
 - collaboration template 3, 198
 - 1_EXCLUDE_VALUES 469
 - 1_FAIL_ON_INVALID_VALUE 469
 - 1_FILTER_ATTRIBUTE 470
 - 1_INCLUDE_VALUES 471
 - adding support for long-lived business process 84
 - ADDITIONAL_RETRIEVE 471
 - Additional_Retrieve process 467
 - coding recommendations 35
 - CollaborationFoundation template 29
 - compiling 6, 101
 - configuration properties 91
 - CONVERT_CREATE 472

- collaboration template (*continued*)
 - CONVERT_UPDATE 472
 - converting 102
 - creating 82
 - CustomerPartnerWrapper template 50
 - declaring variables for 90
 - defined 3, 479
 - deleting 107
 - description of 83
 - design considerations for parallel execution 55
 - editing 10
 - email process for error handling 467
 - exporting BPEL and UML files 102
 - filtering process 465
 - handling dependent data in 39
 - importing BPEL and UML files 102
 - importing classes into 87
 - INFORMATIONAL_EXCEPTIONS 473
 - internationalizing 63
 - ItemWrapper template 51
 - message file 43, 193, 194, 231, 234
 - naming 35, 82
 - opening from a .cwt file 20
 - ports 94
 - properties of 17, 22, 83
 - retrieve process 461
 - saving as a .cwt file 20
 - scenarios 96
 - SEND_EMAIL 473
 - specifying a package for 86
 - specifying the minimum transaction level 84
 - standard 29
 - standard process 461
 - standard properties 468
 - testing 107
 - updating old templates 103
 - USE_RETRIEVE 473
 - Use_Retrieve process 462
- collaboration template function block 273
 - AnyException 274
 - AttributeException 274
 - Get Message 274
 - Get Message with Parameter 275
 - Get Name 275
 - Get Property 276
 - Get Property Array 276
 - Implicit DB Bracketing 276
 - Is Trace Enabled 277
 - JavaException 277
 - ObjectException 277
 - OperationException 278
 - Property Exists 278
 - Raise Collaboration Exception 278
 - Raise Collaboration Exception 1 279
 - Raise Collaboration Exception 2 280
 - Raise Collaboration Exception 3 280
 - Raise Collaboration Exception 4 281
 - Raise Collaboration Exception 5 282
 - Raise Collaboration Exception with Parameter 282
 - Send Email 283
 - ServiceCallException 284
 - summary 273
 - SystemException 284
 - TransactionException 284
- collaboration variable
 - naming 35
 - populating 203
- collaboration variable (*continued*)
 - user-defined 198
- collaboration-generated trace message 195
- collaboration-specific property 92
- CollaborationException class 11, 167, 423, 427, 429
 - currentException and 167
 - defined 423, 479
 - deprecated methods 427
 - getMessage() 423
 - getMsgNumber() 424
 - getSubType() 424
 - getText() 427
 - getType() 425
 - method summary 423
 - toString() 426
- CollaborationFoundation template 29
 - data access 30
 - data piping 30
 - delegating dependent data checking 40
 - error handling 33
 - extending 35
 - features 30
 - handling dependent data 39
 - handling triggering business objects 30
 - ports 34
 - standard properties 31, 468
 - synchronization 29
 - using 33
- collaborationinstancecachesize collaboration configuration property 199
- collaborations for bidirectional languages, Designing 69
- color of symbols and links
 - changing 189
- Commit 286
- commit() 452
- commit() method 227, 400
- comparing
 - business object arrays 387
 - business object attribute values 207, 370, 371
 - key attribute values 369
- compensation 131, 143
 - common types of 131
 - defined 131, 479
 - defining 131
- Compile Output window 16, 21
- compiling
 - collaboration templates 6
 - compiling a single template 101
 - compiling multiple templates 101
 - files created during compilation 101
 - resolving compilation errors 101
- Component groups 156
 - concurrent processing 55
 - design considerations for 55
 - ensuring data consistency via event isolation 56
 - ensuring data consistency via event sequencing 56
 - problems in 55
- condition 124
- Condition 334
- Condition Editor 122
- configuration parameter
 - LOG_FILE system 354, 364
 - MAX_LOG_FILE_SIZE system 354, 364
 - MIRROR_LOG_TO_STDOUT system 354, 364
 - NUMBER_OF_ARCHIVE_LOGS system 354, 364
- configuration property
 - collaborationinstancecachesize collaboration 199

- configuration property (*continued*)
 - enableinstancereuse collaboration 199
- Configurator, The BiDiTransformation parameter in the Connector 71
- Configuring database connection resilience feature 72
- connection
 - determining if active 228, 407
 - obtaining 213, 347
 - opening 213
 - releasing 228, 408
 - transaction programming model 224, 225, 291, 348
- Connection links 155
- connection pool 213, 228, 291, 292, 348, 408
- connection resilience feature, Customizing database 73
- connection resilience feature, Default settings for database 74
- Connector Configurator, The BiDiTransformation parameter in the 71
- connectors for bidirectional languages, Enabling 70
- constant
 - MAPCONTEXT 415
 - PARAM_IN 299, 413
 - PARAM_INOUT 299, 413
 - PARAM_OUT 299, 413
- constant, param_in 223
- constant, param_out 223
- constructor
 - cwdbstoredprocedureparam() 222
 - cxexecutioncontext() 202
 - CxExecutionContext() 415
- context
 - execution 415, 417
- Context menu 153
- control flow
 - branching 38
 - defined 479
- CONVERT_CREATE property 472
- CONVERT_UPDATE property 472
- Copy 238
- copy() method 36, 206, 368, 384
- copying 206
 - attribute values 209
 - business object 368
 - business object variables 36
- correlation attribute
 - defined 132, 479
 - initializing 132
 - matching 133
 - requirements for using 132
 - setting 133
- Create request 92, 97, 131
- creating a collaboration property for the map name 200
- creating a message file 231
- currentException system-generated variable 91, 167, 173, 479
- CustomerPartnerWrapper template 50
- Customizing database connection resilience feature 73
- CwBidiEngine class 419
- CwBidiEngine method summary 419
- cwdbconnection class
 - creating object 213
 - methods for calling stored procedures 220
 - methods for row access 214
 - methods for transaction management 227
- CwDBConnection class 399, 409
 - beginTransaction() 399
 - commit() 400
 - creating object of 347
 - executePreparedSQL() 401

- CwDBConnection class (*continued*)
 - executeSQL() 402
 - executeStoredProcedure() 404
 - getUpdateCount() 405
 - hasMoreRows() 406
 - inTransaction() 406
 - isActive() 407
 - method summary 399
 - nextRow() 407
 - release() 408
 - rollback() 409
- cwdbstoredprocedureparam class 222
- CwDBStoredProcedureParam class 411, 413
 - constructor 411
 - getParamType() 412
 - getValue() 413
 - method summary 411
- cwdbstoredprocedureparam() constructor 222
- CwDBStoredProcedureParam() constructor 411
- cwdbtransactionexception exception 225, 228, 229
- CwDBTransactionException exception 286, 296, 349, 400, 401, 408, 409
- CxExecutinoContext class
 - MAPCONTEXT 415
- cxexecutioncontext class 202
- CxExecutionContext class 415, 417
 - CxExecutionContext() 415
 - defined 415
 - getContext() 416
 - setContext() 416
- cxexecutioncontext() constructor 202
- CxExecutionContext() constructor 415

D

- data
 - dependent 39
- data access 30
- data filtering 32, 465
 - using standard properties for 32
- data piping 30
- data structure
 - EventKeyAttrDef() 430
 - EventQueryDef() 431
 - FailedEventInfo() 433
- data type
 - boolean 208, 210, 224
 - date 224
 - double 208, 210, 224
 - float 208, 210, 224
 - int 208, 210, 224
 - long 208, 210, 224, 411
 - longtext 208
 - string 208, 210, 224
- database
 - connecting to 213, 228, 347
 - executing a query in 213, 287, 288, 290, 402, 403, 404
 - handling data from 214
 - modifying 217, 218
 - querying 214, 218, 406, 407
 - rows affected by last write 405
- database connection function block 285
- Database connection function block
 - Begin Transaction 285
 - Commit 286
 - Execute Prepared SQL 287
 - Execute Prepared SQL with Parameter 288

- Database connection function block *(continued)*
 - Execute SQL 288
 - Execute SQL with Parameter 289
 - Execute Stored Procedure 290
 - Get Database Connection 291
 - Get Database Connection with Transaction 291
 - Get Next Row 292
 - Get Update Count 293
 - Has More Rows 294
 - In Transaction 294
 - Is Active 295
 - Release 295
 - Roll Back 296
- database connection resilience feature, configuring 72
- database connection resilience feature, Customizing 73
- database connection resilience feature, Default settings for 74
- Database Resilience, System Manager showing default values for 74
- database stored procedure function block 299
 - Get Param Type 299
 - Get Param Value 300
 - New DB Stored Procedure Param 300
- Date After 312
- Date Before 312
- Date class 411
- date data type 224
- Date data type 90
- Date Equals 313
- date function block
 - Add Day 311
 - Add Month 311
 - Add Year 312
 - Date After 312
 - Date Before 312
 - Date Equals 313
 - Format Change 313
 - Get Day 313
 - Get Month 313
 - Get Year 314
 - Get Year Month Day 314
 - Now 314
 - yyyy-MM-dd 314
 - yyyyMMdd 315
 - yyyyMMdd HH mm 315
- Date function block 311
- decision node 120
 - adding to activity diagram 122
 - creating a default branch 123
 - creating a normal branch 122
 - creating an exception branch 123
 - default branch 120
 - defined 120, 480
 - defining branches and conditions 121
 - exception branch 120
 - normal branch 120
 - number of permitted links 117
 - types of branches in 120
 - using Condition Editor to define conditions 122
- decision properties dialog box 121
- declaration
 - defined 480
- default branch
 - creating 123
 - defined 120
- Default settings for database connection resilience feature 74
- default values for Database Resilience, System Manager showing 74
- definitions
 - Activity 154
 - tags for activity 156
- delegation 58
- Delete request 97, 131
- delete statement 217
- DELETE statement 287, 289, 402, 403
- deleteBO() 452
- deleteState() 453
- dependent data 39
 - creating scenarios to delegate checking for 43
 - delegating the checking of 40
 - handling in business objects 39
 - recursively checking 44
- deprecated method
 - BusObj class 384
 - CollaborationException class 427
- Designing collaborations for bidirectional languages 69
- destination
 - log 354, 364
- development process 12
- diagram
 - main activity 134, 135, 143, 169, 170
 - parent 110, 135, 137, 143, 144, 145
 - successfully ending the main 170
- diagram display, changing 188
- diagram editor 18, 19, 186
 - displaying 109
 - selecting and deselecting symbols in 109
 - using context menus 109
- dialog
 - Action Properties 113
 - Decision Properties 121
 - End Failure Properties 144
 - End Success Properties 144
 - grid properties 186
 - Iterator Properties 140
 - Link Properties 118
 - Service Call Properties 128
 - Subdiagram Properties 136
 - symbol properties 191
 - Symbol Properties 110, 112
- dialog, Bidirectional language parameter input 72
- display
 - changing diagram 188
 - changing general 187
- doAgg() 446
- doMergeHash() 446
- doRecursiveAgg() 447
- doRecursiveSplit() 447
- Double class
 - as stored-procedure parameter type 411
- double data type 90, 208, 210, 224, 373, 379, 383, 388
 - as stored-procedure parameter type 411
- dropFailedEvent() 343
- Duplicate 239, 262
- duplicate() method 36, 206, 207, 369, 386
- duplicating
 - business object 207, 369
 - business object array 386
- dynamic service calls 54
- dynamicSend() 344

E

- Edit menu 21, 152
- editor
 - diagram 186
- Editor
 - Activity 114
 - Condition 122
- Editor keyboard shortcuts, Activity 153
- Editor menu
 - Activity 151
- elementAt() method 387
- email notification 283, 354, 363
- email process for error handling
 - collaboration template 467
- enableinstancereuse collaboration configuration property 199
- enabling a collaboration to save failed events 77
- Enabling connectors for bidirectional languages 70
- encoding, character 64
- End Failure Properties dialog 144
- End Failure symbol
 - adding to activity diagram 144
 - defining 144
 - description 145
 - End Failure Properties dialog for 144
 - in subdiagram 137
 - label for 145
 - properties of 144
- End Success Properties dialog 144
- End Success symbol
 - adding to activity diagram 143
 - defining 144
 - description 144
 - End Success Properties dialog for 144
 - in subdiagram 137, 138
 - label for 144
 - properties of 144
- ending a subdiagram or iterator
 - successfully 171
- ending the main diagram
 - successfully 170
- English 334
- enumeration class 214
- environment variable
 - CLASSPATH 86, 89, 166
 - JCLASSES 165
- Environment variable
 - CLASSPATH 166
 - JCLASSES 165
- Equal Keys 239
- equalKeys() method 369
- Equals 240, 262
- equals() method 115, 197, 207, 370, 387
- equalsShallow() method 371
- error handling 33
 - during delegation of dependent data checking 41
 - FAIL_ON_CONTACT_ERROR 41
 - FIND_ALL_ITEM_ERRORS 41
 - using collaboration properties 41
- error message 101, 194, 353
- event
 - coding to save a failed 76
 - enabling a collaboration to save failed 77
 - handling failed 75
 - resubmitting failed 78
 - saving a failed 75, 78
- event isolation 56, 57, 61, 480
 - design rules for 58
 - event isolation (*continued*)
 - handling child business objects as reference-valued 60
 - using delegation 58
- event sequencing 56, 480
 - controlling 56
- EventKeyAttrDef() 430
- EventManager package
 - classes 429
- EventQueryDef() 431
 - static constants 431
- exception
 - AnyException 168, 173, 358
 - AttributeException 168, 358
 - catching 172
 - categories of 171
 - class for 11, 423
 - cwdbtransactionexception 225, 228, 229
 - CwDBTransactionException 286, 296, 349, 400, 401, 408, 409
 - defined 167, 480
 - formatting 426
 - handling boot-time recovery transport-related 180
 - handling particular service-call 178
 - handling runtime transport-related 179
 - how to handle 171
 - howprocessed 168
 - JavaException 168, 172, 358, 424, 426
 - message number 424
 - not catching 172
 - ObjectException 168, 358, 426
 - OperationException 168, 358, 426
 - raising 37, 174, 357
 - ServiceCallException 126, 134, 168, 178, 358, 424, 426
 - subtypes of 424
 - SystemException 168, 358, 426
 - text 423, 426
 - TransactionException 168, 358, 426
 - types of 425, 426
- exception branch
 - creating 123
 - defined 120
- exception function block 303
 - Catch Collaboration Exception 303
 - Get Message 303
 - Get Message Number 304
 - Get Subtype 304
 - Get Type 305
 - To String 306
- exception handling 37, 137, 138, 181
- exception object 167
 - contents of 167
 - exception subtype 167
 - exception text 426
 - exception type 167, 168, 426
 - message 167, 423
 - message number 167, 424
- exception state, processing the 169
- exception subtype
 - AppRequestNotYetSent 181
 - AppUnknown 179
 - ServiceCallTransportException 179
- Exceptions from the Collaboration API 181
- Execute Prepared SQL 287
- Execute Prepared SQL with Parameter 288
- Execute SQL 288
- Execute SQL with Parameter 289
- Execute Stored Procedure 290

- executePreparedSQL() method 217, 221
- executePreparedSQL() method 401
- executesql() method 213, 220
- executeSQL() method 402
- executestoredprocedure() method 220, 221
- executeStoredProcedure() method 404
- execution context 415, 417
- execution function block 309
 - Get Context 309
 - MAPCONTEXT 309
 - New Execution Context 309
 - Set Context 310
- execution path 5, 143
 - choosing 124
 - of main diagram 136
 - of subdiagram 136
 - terminating in failure 144
 - terminating in success 143
 - using property to choose 92
- execution status
 - failure 144
 - successful 143
- Exists 240
- exists() method 372
- existsConfigProperty() method 345
- explanations
 - reading message 233
- explicit transaction bracketing
 - releasing the connection 228

F

- failed event
 - coding to save 76
 - enabling a collaboration to save 77
 - handling 75
 - required data structures 78
 - resubmitting 78
 - saving 78
 - saving a 75
- FailedEventInfo() 433
 - static constants 433
- failure execution status 144
- feature, Customizing database connection resilience 73
- feature, Default settings for database connection resilience 74
- file
 - creating a message 231
 - InterchangeSystem.log log 354, 364
 - start_server.bat 166
- File menu 20, 151
- filter class 435
- filter method
 - Filter() 436
 - filterExcludes() 437
 - filterIncludes() 438
 - recurseFilter() 439
 - recursePreReqs() 440
- filter method summary 435
- Filter method summary 435
- Filter() 436
- filterExcludes() 437
- filterIncludes() 438
- filtering process
 - collaboration template 465
- filtering, data 32
- Float class
 - as stored-procedure parameter type 411
- float data type 90, 208, 210, 224, 373, 379, 383, 388
 - as stored-procedure parameter type 411
- flow
 - unresolved 138, 170
- flow locale 67
- Flow Manager tool 170
- flow trigger 9
 - assigning to scenario 97
 - copying 205
 - handling flow triggers in scenarios 97
 - processing 35
 - receiving 94
 - variable for 205
 - when deleted 96
- Format Change 313
- French 334
- full-valued business object 60
 - defined 480
- function block 154
 - adding custom Jar libraries as 162
 - Business object 237
 - Business object array 261
 - collaboration template 273
 - Copy 238
 - customizing Jar library properties 163
 - database connection 285
 - database stored procedure 299
 - Duplicate 239
 - Equal Keys 239
 - Equals 240
 - Exists 240
 - Get Boolean 241
 - Get Business Object 241
 - Get Business Object Array 242
 - Get Business Object Type 243
 - Get BusObj At 243
 - Get Double 243
 - Get Float 244
 - Get Int 245
 - Get Locale 245
 - Get Long 246
 - Get Long Text 246
 - Get Object 247
 - Get String 248
 - Get Verb 248
 - Is Blank 249
 - Is Business Object 249
 - Is Key 250
 - Is Null 250
 - Is Required 251
 - Iterate Children 251
 - Keys to String 251
 - New Business Object 252
 - New Business Object Array 252
 - New Constant 156
 - organization of 157
 - Output 241
 - Set BusObj At 253
 - Set Content 253
 - Set Default Attribute Values 253
 - Set Keys 254
 - Set Locale 254
 - Set Value 254
 - Set Value By Position 255
 - Set Value with Create 256
 - Set Verb 256
 - Set Verb with Create 257

- function block (*continued*)
 - Shallow Equals 257
 - Size 258
 - summary of collaboration template 273
 - summary of database stored procedure 299
 - supported 157
 - To String 258
 - Valid Data 258
 - Verb
 - Create 259
 - Delete 259
 - Retrieve 260
 - Update 260
 - web services 155

G

- general display, changing 187
- German 335
- Get Boolean 241
- Get Business Object 241
- Get Business Object Array 242
- Get Business Object Type 243
- Get BusObj At 243
- Get Context 309
- Get Country 335
- Get Database Connection 291
- Get Database Connection with Transaction 291
- Get Day 313
- Get Double 243
- Get Element 335
- Get Element At 263
- Get Elements 263
- Get Float 244
- Get Language 335
- Get Last Index 263
- Get Locale 245, 274
- Get Long 246
- Get Long Text 246
- Get Message 274, 303
- Get Message Number 304
- Get Message with Parameter 275
- Get Month 313
- Get Name 275
- Get Next Row 292
- Get Object 247
- Get Param Type 299
- Get Param Value 300
- Get Property 276
- Get Property Array 276
- Get String 248
- Get Subtype 304
- Get Type 305
- Get Update Count 293
- Get Verb 248
- Get Year 314
- Get Year Month Day 314
- get() method 208, 372
- getboolean() method 208
- getBoolean() method 372
- getbusobj() method 208, 210
- getBusObj() method 372
- getbusobjarray() method 208, 211
- getBusObjArray() method 372
- getConfigproperty() method 197, 200
- getConfigProperty() method 143, 346
- getConfigPropertyArray() method 197

- getConfigPropertyArray() method 143, 346
- getContext() method 416
- getCount() method (deprecated) 384
- getCurrentLoopIndex() method (Base Collaboration) 347
- getdbconnection() method 213, 225
- getDBConnection() method 347, 348
- getdouble() method 208
- getDouble() method 372
- getElements() method 387
- getfloat() method 208
- getFloat() method 372
- getInt() method 208
- getInt() method 372
- getKeys() method (deprecated) 384
- getKeyValues() 448
- getLastIndex() method 388
- getLocale() method (Base Collaboration) 349
- getLocale() method (BaseCollaboration) 67
- getLocale() method (BusObj) 68, 374
- getlong() method 208
- getLong() method 372
- getlongtext() method 208
- getLongText() method 372
- getMessage() method 167
- getMessage() method (BaseCollaboration) 66, 350
- getMessage() method (CollaborationException) 423
- getMsgNumber() method 167, 424
- getName() method 351
- getParamtype() method 222
- getParamType() method 412
- getString() method 208
- getString() method 372
- getSubType() method 167, 179, 181, 424
- getText() method (deprecated) 427
- gettype() method 206
- getType() method 167, 375, 425
- getupdatecount() method 217
- getUpdateCount() method 405
- getvalue() method 222
- getValue() method 413
- getValues() method (deprecated) 384
- getVerb() method 375
- globals class 441
- globals class method summary 441
- globals method
 - callMap() 443
 - globals() 442
- Globals() 442
- grid properties dialog 186
- group
 - collaboration 52
- groups
 - component 156

H

- handling
 - boot-time recovery transport-related exceptions 180
 - error 33
 - exception 37, 137, 138, 181
 - particular service-call exceptions 178
 - runtime transport-related exceptions 179
- handling failed events 75
- Has More Rows 294
- hasmorerows() method 214, 220
- hasMoreRows() method 406
- Help menu 153

- hierarchical business object
 - coding techniques 210
 - comparing all 370
 - comparing top-level 371
- how exceptions are processed 168
- how to handle exceptions 171

I

- If 326
- Implicit DB Bracketing 276
- implicit transaction bracketing
 - releasing the connection 228
- implicitdbtransactionbracketing() method 225
- implicitDBTransactionBracketing() method 351
- import statement 87, 480
- importing third-party classes to InterChange Server 165
- in parameter 221, 222, 223
- IN parameter 299, 413
- In Transaction 294
- inbound service call
 - asynchronous 127
- including web services 53
- informational message 193, 195, 353
- INFORMATIONAL_EXCEPTIONS property 473
- initializing the collaboration 200
- inout parameter 222
- INOUT parameter 299, 413
- input dialog, Bidirectional language parameter 72
- insert statement 217
- INSERT statement 287, 289, 293, 402, 403, 405
- int data type 90, 208, 210, 224, 373, 379, 383, 388
 - as stored-procedure parameter type 411
- Integer class
 - as stored-procedure parameter type 411
- Integration Component Library user project 82
- InterChange Server
 - importing third-party classes to 165
- InterchangeSystem.log log file 354, 364
- internationalization
 - character-encoding design principles 69
 - considerations for collaboration properties 68
 - defined 63
 - handling email messages 66
 - handling hardcoded strings 66
 - locale 64
 - locale-sensitive design 64
 - obtaining exception messages 66
 - obtaining log messages 66
 - of collaboration template 63
 - of collaboration text strings 65
 - using collaboration message file for text strings 65
- intransaction() method 227
- inTransaction() method 406
- Is Active 295
- Is Blank 249
- Is Business Object 249
- Is Business Object Array 264
- Is Empty 326
- Is Key 250
- Is Null 250
- Is NULL 326
- Is Required 251
- Is Trace Enabled 277
- isactive() method 228
- isActive() method 407
- isBlank() method 376

- isCallerInRole() 352
- isKey() method 376
- isNull() method 377
- isolation
 - event 56, 57, 61
- isRequired() method 378
- isTraceEnabled() method 352
- Italian 336
- Item business object
 - Item 43
 - ItemBasic 43
 - ItemOrder 43
 - ItemPlanning 43
 - prerequisite Item objects 45
 - using ITEM_TYPE property 43
- ItemWrapper template 51
- Iterate Children 251
- Iterate Vector 336
- iterator
 - creating 139
 - defined 139, 480
 - defining 140
 - Iterator Properties dialog 140
 - restrictions on links 117
 - successfully ending 171
 - symbol 139
 - uses for 139
- Iterator Properties dialog 140

J

- Japanese 336
- Jar libraries
 - customizing display settings 163
 - importing as function blocks 162
- java class
 - enumeration 214
 - importing 200
 - java.sql.types 224
 - object 208, 210
 - vector 214, 221
- Java class
 - importing 87
 - importing from third-party packages 88
 - Object 372, 373, 379, 383
 - resolving import errors 89
 - Vector 301, 403, 407, 412
- Java operator 115
 - AND 124
 - arithmetic 116
 - assignment 116
 - equals() 115
 - NOT 384
 - relational 115
- java.lang package 87
- java.sql.types class 224
- java.util package 87, 214
- JavaException 277
- JavaException exception 168, 172, 358, 424, 426
- JCLASSES environment variable 165
- JDK 87

K

- key attribute value
 - checking for 376

- key attribute value *(continued)*
 - comparing 369
 - retrieving as string 378
 - setting 380
- keyboard shortcuts
 - Activity Editor 153
- Keys to String 251
- keysToString() method 378, 384
- Korean 336

L

- label
 - for End Failure 145
 - for End Success 144
 - for service call 128
 - for subdiagram 137
 - for symbols 112
 - for transition link 118
 - viewing 22
- language parameter input dialog, Bidirectional 72
- languages, Designing collaborations for bidirectional 69
- languages, Enabling connectors for bidirectional 70
- Left Fill 326
- Left String 327
- level
 - trace 195, 197, 352
- link
 - connection 155
 - transition 116
- Link Properties dialog 118
- locale
 - business object 67
 - collaboration 67, 349
 - defined 63
 - design considerations for localized collaborations 64
 - design considerations for text strings 65
 - flow 67
 - information provided by 64
- log destination 354, 364
- Log error 317
- Log Error ID 317
- Log Error ID 1 318
- Log Error ID 2 318
- Log Error ID 3 318
- log file
 - InterchangeSystem.log 354, 364
- Log Information 319
- Log Information ID 319
- Log Information ID 1 319
- Log Information ID 2 319
- Log Information ID 3 320
- Log Warning 320
- Log Warning ID 320
- Log Warning ID 1 320
- Log Warning ID 2 321
- Log Warning ID 3 321
- LOG_FILE system configuration parameter 354, 364
- logerror() method 193, 194, 231
- logError() method 65, 174, 353
- logging 65, 193
 - example 194
 - levels 194
 - principles 194
 - severity levels 193
- logging and tracing function block 317
 - Log error 317

- logging and tracing function block *(continued)*
 - Log Error ID 317
 - Log Error ID 1 318
 - Log Error ID 2 318
 - Log Error ID 3 318
 - Log Information 319
 - Log Information ID 319
 - Log Information ID 1 319
 - Log Information ID 2 319
 - Log Information ID 3 320
 - Log Warning 320
 - Log Warning ID 320
 - Log Warning ID 1 320
 - Log Warning ID 2 321
 - Log Warning ID 3 321
 - Trace 321
 - Trace ID 1 321
 - Trace ID 2 322
 - Trace ID 3 322
 - Trace on Level 323
- logical operator 124, 384
- loginfo() method 193, 195, 231
- logInfo() method 65, 353
- logwarning() method 193, 231
- logWarning() method 65, 353
- long data type 90, 208, 210, 224, 373, 379, 383, 411
- long-lived business process
 - adding support for 84
 - defined 8, 480
 - design considerations 53
 - special considerations for collaboration groups 52
 - specifying service call timeout values 130
 - using a collaboration as 8
 - using collaboration configuration properties with 94
 - using correlation attributes with 132
 - using dynamic timeout values 94
 - using scenario variables with 100
 - using template variables with 90
- LongText class
 - setting attribute 255, 379
- longtext data type 208
- LongText data type 90, 373, 388
- Loop 337
- Lower Case 327

M

- main activity diagram 134, 135, 143, 169, 170
- main diagram
 - successfully ending the 170
- Manager showing default values for Database Resilience, System 74
- map 199, 415
 - calling 200
- map name, creating a collaboration property 200
- MAPCONTEXT 309
- MAPCONTEXT constant 415
- matching
 - port 57, 62
- Max Attribute Value 264
- Max Business Object Array 265
- Max Business Objects 266
- MAX_LOG_FILE_SIZE system configuration parameter 354, 364
- max() method 388
- maxBusObjArray() method 389
- maxBusObjs() method 390

- menu
 - Activity Editor 151
 - Context 153
 - Edit 152
 - File 151
 - Help 153
 - Tools 153
 - View 152
- menus.
 - See* Process Designer menus
- merge() 448
- message
 - collaboration-generated trace 195
 - error 194, 353
 - informational 193, 195, 353
 - parameters in 233
 - revising 194
 - severity 194
 - system-generated trace 195, 197
 - trace 195, 197
 - warning 193, 353
- message explanations
 - reading 233
- message file 7
 - defined 193
 - explanations 233
 - localized 65
 - location of 7, 232
 - maintaining 234
 - name of 232
 - operations that use 231
 - setting up 231, 234
 - using 65, 193, 194
 - using with synchronization and wrapper
 - collaborations 43
- message file, creating a 231
- Message logging 423
- method
 - begintransaction() 227
 - commit() 227
 - copy() 36, 206
 - dropFailedEvent() 343
 - duplicate() 36, 206, 207
 - dyanmicSend() 344
 - equals() 115, 197, 207
 - executepreparedsql() 217, 221
 - executesql() 213, 220
 - executestoredprocedure() 220, 221
 - existsConfigProperty() 345
 - get() 208
 - getboolean() 208
 - getbusobj() 208, 210
 - getbusobjarray() 208, 211
 - getConfigProperty() 197, 200
 - getConfigProperty() 143, 346
 - getConfigPropertyarray() 197
 - getConfigPropertyArray() 143, 346
 - getContext() 416
 - getdbconnection() 213, 225
 - getDBCconnection() 347, 348
 - getdouble() 208
 - getfloat() 208
 - getInt() 208
 - getlong() 208
 - getlongtext() 208
 - getMessage() 167
 - getMessage() (CollaborationException) 423
- method (*continued*)
 - getMsgNumber() 167, 424
 - getName() 351
 - getParamtype() 222
 - getParamType() 412
 - getString() 208
 - getSubType() 167, 179, 181, 424
 - getType() 206
 - getType() 167, 425
 - getupdatecount() 217
 - getValue() 222
 - getValue() 413
 - hasmorerows() 214, 220
 - implicitdbtransactionbracketing() 225
 - implicitDBTransactionBracketing() 351
 - intransaction() 227
 - inTransaction() 406, 407
 - isactive() 228
 - isCallerInRole() 352
 - isTraceEnabled() 352
 - logerror() 193, 194, 231
 - logError() 65, 174, 353
 - loginfo() 193, 195, 231
 - logInfo() 65, 353
 - logwarning() 193, 231
 - logWarning() 65, 353
 - nextrow() 214, 220
 - nextRow() 407
 - queryFailedEvents 355
 - rai
 - See* xception()
 - release() 408
 - resubmitFailedEvent() 360
 - rollback() 227
 - rollBack() 409
 - saveFailedEvent() 362
 - sendEmail() 66, 362
 - sendMail() 65
 - set() 204, 207, 210
 - setcontext() 202
 - setContext() 416
 - setLocale() 68
 - toString 426
 - toString() 167, 427
 - trace() 195, 231, 363
- method (Base Collaboration)
 - getCurrentLoopIndex() 347
 - getLocale() 349
- method (BaseCollaboration)
 - getMessage() 350
- method (BaseCollaboration), getLocale() 67
- method (BaseCollaboration), getMessage() 66
- method (BusObj), getLocale() 68
- method getText() (deprecated) 427
- method summary, CwBidiEngine 419
- Min Attribute Value 267
- Min Business Object Array 267
- Min Business Objects 268
- min() method 391
- minBusObjArray() method 392
- minBusObjs() method 393
- Minimal-effort transaction level 85
- minimum transaction level 84
- MIRROR_LOG_TO_STDOUT system configuration
 - parameter 354, 364
- Move Attribute in Child 337
- multithreading 55

multithreading (*continued*)
use of event sequencing 56

N

name
collaboration configuration property 93
for action node 113
of transition link 118
port 95
template 82

naming conventions
for collaboration configuration properties 93
for collaboration properties 35
for collaboration templates 35, 82
for collaboration variables 35
for ports 95
for scenario 97
for scenarios 97

New Business Object 252
New Business Object Array 252
New Constant function block 156
New DB Stored Procedure Param 300
New Execution Context 309
New Locale 337
New Locale with Language 338
New Vector 338

nextrow() method 214, 220
nextRow() method 407

node
decision 120
defined 111
types of 111

normal branch
creating 122
defined 120
using Condition Editor to define conditions 122

normal state
processing 169

Not catching exception 172
NOT operator 384

notification
email 354, 363

Now 314
Nudge toolbar 21, 23
null attribute value 207, 212, 377
NUMBER_OF_ARCHIVE_LOGS system configuration
parameter 354, 364

O

object
exception 167
full-valued business 60

object class 208, 210
Object class 372, 373, 379, 383
Object to String 327
ObjectException 277
ObjectException exception 168, 358, 426
OperationException 278
OperationException exception 168, 358, 426

operator
AND 124
arithmetic 116
assignment 116
Java 115

operator (*continued*)
logical 124
relational 115

optimization 134
Organization of function blocks 157
out parameter 220, 222, 223
OUT parameter 299, 300, 413
outbound service call, asynchronous 127
Output 241
Output window 101

P

package 87, 214
defined 86, 480
java.util 87, 214
specifying for a collaboration template 86

Package
importing Java packages 162

param_in constant 223
PARAM_IN constant 299, 413
PARAM_INOUT constant 299, 413
param_out constant 223
PARAM_OUT constant 299, 413

paramete
OUT 300

parameter
in 221, 222, 223
IN 299, 413
inout 222
INOUT 299, 413
LOG_FILE system configuration 354, 364
MAX_LOG_FILE_SIZE system configuration 354, 364
MIRROR_LOG_TO_STDOUT system configuration 354,
364
NUMBER_OF_ARCHIVE_LOGS system
configuration 354, 364
out 220, 222, 223
OUT 299, 413

parameter in the Connector Configurator, The
BiDiTransformation 71

parameter input dialog, Bidirectional language 72

parent diagram 110, 135, 137, 143, 144, 145

path
execution 143

performance 134

persistBO() 453

pipng
data 30

pool
connection 213, 228, 291, 292, 348

populating the collaboration variable 203

port 94
adding ports for dependent business objects 40
CollaborationFoundation's DestinationAppRetrieve
port 34
CollaborationFoundation's From port 35
CollaborationFoundation's To port 34
creating 95
defined 94, 480
deleting 96
external 9
for flow trigger 98
for triggering event 98
in CollaborationFoundation template 34
internal 9
matching 57

- port (*continued*)
 - naming 95
 - renaming 96
 - type of 96
 - using a Port connector 96
 - variable 205
 - variable for 91, 96
 - WrapperFoundation ports 48
 - WrapperFoundation's DestinationAppRetrieve port 49
 - WrapperFoundation's From port 49
 - WrapperFoundation's SourceApp port 49
 - WrapperFoundation's To port 50
- Port connector 96
- port matching 57, 62
 - example of matching ports 57
 - example of non-matching ports 58
- Proceeding with scenario logic 173
- Process Designer
 - Activity Editor 114
 - Compile Output window 16, 21
 - customizing the layout of 23
 - diagram editor 18, 19, 109
 - layout of 16
 - main window 23
 - menus of 109
 - menus.
 - See* Process Designer menus
 - output window 101
 - starting 15
 - status bar 22
 - Template Definitions window 17
 - template tree view 16, 21
 - toolbars.
 - See* Process Designer toolbars
- Process Designer menus
 - descriptions of 20
 - Edit menu 21
 - File menu 20
 - Template menu 22
 - View menu 21
 - Window menu 23
- Process Designer toolbars
 - Alignment toolbar 21, 23
 - displaying 21, 24
 - Nudge toolbar 21, 23
 - overview of 23
 - Standard toolbar 21, 23
 - Symbols toolbar 21, 23, 110
 - Zoom/Pan toolbar 21, 23
- processin
 - concurrent 55
- processing of business objects
 - recursive 44, 45
- Processing the Exception state 169
- Processing the Normal state 169
- properties dialog
 - grid 186
 - service call 128
 - symbol 191
- Properties dialog
 - Action 113
 - Decision 121
 - End Failure 144
 - End Success 144
 - Iterator 140
 - Link 118
 - Service Call 128

- Properties dialog (*continued*)
 - Subdiagram 136
 - Symbol 110, 112
- property
 - collaborationinstancecachesize collaboration configuration 199
 - enableinstancereuse collaboration configuration 199
- Property Exists 278

Q

- query
 - sql 212, 229
- queryFailedEvents() 355

R

- rai
 - See* xception() method
- Raise Collaboration Exception 278
- Raise Collaboration Exception 1 279
- Raise Collaboration Exception 2 280
- Raise Collaboration Exception 3 280
- Raise Collaboration Exception 4 281
- Raise Collaboration Exception 5 282
- Raise Collaboration Exception with Parameter 282
- Raise Error 338
- Raise Error Type 338
- raising the exception 174
- reading message explanations 233
- recoverBO() 454
- recurseFilter() 439
- recursePreReqs() 440
- recursive processing of business objects 44, 45
 - InstalledProductSync collaboration 46
 - ItemSync collaboration 46
- reference-valued business object 60
 - defined 480
- referenced business object
 - synchronizing 40
 - verifying 40
- relational operator 115
- Release 295
- release() method 408
- releaseDBCConnection() 455
- Remove All Elements 269
- Remove Element 269
- Remove Element At 270
- removeAllElements() method 394
- removeElement() method 394
- removeElementAt() method 394
- removing
 - all elements of a business object array 394
 - element of a business object array 394
- Repeat 327
- Replace 328
- request 112, 126, 131, 171, 423
 - Create 131
 - Delete 131
 - Retrieve 131
 - unsent service call 181
 - Update 131
- resetData() 455
- resilience feature, Customizing database connection 73
- resilience feature, Default settings for database connection 74

- Resilience, System Manager showing default values for
 - Database 74
- resubmitFailedEvent() 360
- resubmitting failed event 78
- Retrieve process
 - collaboration template 461
- Retrieve request 131, 424
- Retrieve verb
 - behavior of 30
- retrieveState() 455
- retrieving
 - business object array contents 387
 - business object array maximum value 388, 389, 390
 - business object array minimum value 391, 392, 393
 - business object array values as string 397
 - business object attribute value 208, 372
 - business object from array 387
 - business object key attribute value as string 378
 - business object type 375
 - business object verb 375
 - collaboration object name 351
 - configuration property value 143, 197, 346, 415, 416
 - exception as string 426
 - exception subtype 424
 - exception type 425
 - last index from business object array 388
 - number of elements in business object array 395
 - triggering business object data 30
- return value
 - for service call 134
- Right Fill 328
- Right String 328
- Roll Back 296
- rollback() method 227
- rollback() method 409
- runtime transport-related exceptions, handling 179

S

- save a failed event
 - coding 76
- save failed events
 - enabling a collaboration to 77
- saveFailedEvent() 362
- saveState() 456
- saving a failed event 75, 78
- scenario 4
 - aligning 190
 - assigning a triggering event 97
 - creating 97
 - defined 4, 96, 480
 - defining scenario variables 99
 - deleting 22, 100
 - flow trigger 97, 205
 - for delegating dependent data checking 43
 - handling a flow trigger 97
 - naming 97
 - naming conventions 97
 - triggering event 205
- Scenario Definitions dialog 99
- scenario logic
 - proceeding with 173
- scenario variable
 - defined 99, 480
 - subdiagram and 136
 - using with a long-lived business process 100
- select statement 214, 218

- SELECT statement 287, 289, 293, 294, 402, 403, 406, 408
- Send Email 283
- SEND_EMAIL property 473
- sendEmail() method 66, 362
- sendMail() method 65
- sequencing event
 - controlling 56
- sequencing, event 56
- Server, Importing third-party classes to InterChange 165
- service call
 - as subtransaction step 131
 - asynchronous inbound 127
 - asynchronous inbound service call 127
 - asynchronous outbound 127
 - asynchronous outbound service call 127
 - compensation 131
 - creating 127
 - defined 112, 480
 - defining 128
 - dynamic 54
 - exactly-once requests 178
 - label for 128
 - optional properties for 128
 - overview of 126
 - performance considerations 134
 - relationship between service call and action node 126
 - required properties for 128
 - results of 134
 - return values 134
 - specifying the type of 129
 - specifying timeout value for 130
 - subtransaction steps and 131
 - synchronous 127
 - synchronous service call 127
 - transport-related exceptions 425
 - types of 112, 126
 - unsent 181
 - using correlation attributes with 132
 - using dynamic timeout value for 94
- service call exceptions
 - handling 178
- Service Call Properties dialog 128
- ServiceCallException 284
- ServiceCallException exception 126, 134, 168, 178, 358, 424, 426
- ServiceCallTransportException exception subtype 179, 425
- Set BusObj At 253
- Set Content 253
- Set Context 310
- Set Default Attribute Values 253
- Set Element At 270
- Set Keys 254
- Set Locale 254
- Set Value 254
- Set Value By Position 255
- Set Value with Create 256
- Set Verb 256
- Set Verb with Create 257
- set() method 204, 207, 210, 379, 384
- setcontext() method 202
- setContext() method 416
- setDBConnection() 456
- setDefaultAttrValues() method 380
- setElementAt() method 395
- setKeys() method 380
- setLocale() method 68, 381
- settings for database connection resilience feature, Default 74

- setVerb() method 381, 384
- setWithCreate() method 382
- Shallow Equals 257
- shortcuts
 - Activity Editor keyboard 153
- showing default values for Database Resilience, System Manager 74
- Simplified Chinese 338
- Size 258, 271, 339
- size() method 388, 395
- SmartCollabService class 445
- SmartCollabService method
 - doAgg() 446
 - doMergeHash() 446
 - doRecursiveAgg() 447
 - doRecursiveSplit() 447
 - getKeyValues() 448
 - merge() 448
 - SmartCollabService() 445
 - split() 449
- SmartCollabService method summary 445
- SmartCollabService() 445
- split() 449
- sql query 212, 229
 - checking for more rows 214
 - executing 213
 - prepared 217
 - retrieving next row 214
 - static 213
- SQL query
 - checking for more rows 406
 - executing 401, 402, 404
 - prepared 401
 - retrieving next row 407
 - static 402
- standard collaboration template 29
- standard process
 - collaboration template 461
- standard properties 468
 - 1_EXCLUDE_VALUES 469
 - 1_FAIL_ON_INVALID_VALUE 469
 - 1_FILTER_ATTRIBUTE 470
 - 1_INCLUDE_VALUES 471
 - ADDITIONAL_RETRIEVE 31, 471
 - business flow properties 31
 - collaboration template 468
 - CollaborationFoundation template 31
 - CONVERT_CREATE 31, 472
 - CONVERT_UPDATE 31, 472
 - data filtering properties 32, 466
 - error-handling properties 33
 - INFORMATIONAL_EXCEPTIONS 473
 - SEND_EMAIL 473
 - USE_RETRIEVE 31, 473
- standard property 91
- Standard toolbar 21, 23
- Start symbol
 - initializing correlation attributes in 133
- start_server.bat file 89, 166
- state
 - processing exception 169
 - processing normal 169
- StateManagement class 451
- StateManagement method
 - beginTransaction() 452
 - commit() 452
 - deleteBO() 452
- StateManagement method (*continued*)
 - deleteState() 453
 - persistBO() 453
 - recoverBO() 454
 - releaseDBConnection() 455
 - resetData() 455
 - retrieveState() 455
 - saveState() 456
 - setDBConnection() 456
 - StateManagement() 457
 - updateBO() 457
 - updateState() 457
- StateManagement method summary 451
- StateManagement() 457
- statement
 - call 220, 221
 - CALL 402, 403
 - delete 217
 - DELETE 287, 289, 402, 403
 - insert 217
 - INSERT 287, 289, 293, 402, 403, 405
 - select 214, 218
 - SELECT 287, 289, 293, 294, 402, 403, 406, 408
 - update 217
 - UPDATE 287, 289, 293, 402, 403, 405
- statement, CALL 287, 289
- status
 - failure execution 144
 - successful execution 143
- step
 - subtransaction 131, 143
- stored procedure
 - creating object for parameter 222, 411
 - executing 219, 287, 289, 402, 403, 404
 - in/out parameter type 221, 222, 412
 - parameter 221
 - parameter mapping from java object to jdbc 224
 - parameter value 222, 413
 - query result 220, 293, 294, 406, 408
- String class
 - as stored-procedure parameter type 411
- string data type 208, 210, 224
- String data type 90, 373, 379, 383, 388
- string function block 325
 - Append Text 325
 - If 326
 - Is Empty 326
 - Is NULL 326
 - Left Fill 326
 - Left String 327
 - Lower Case 327
 - Object to String 327
 - Repeat 327
 - Replace 328
 - Right Fill 328
 - Right String 328
 - Substring by Position 328
 - Substring by Value 329
 - Text Equal 329
 - Text Equal Ignore Case 329
 - Text Length 330
 - Trim Left 330
 - Trim Right 330
 - Trim Text 330
 - Upper Case 330
- Stringent transaction level 85
- subdiagram 5, 134

- subdiagram (*continued*)
 - completion status of 137
 - creating 136
 - defined 480
 - defining 136
 - deleting 137
 - description 137
 - label for 137
 - naming 136
 - parent diagram of 135
 - properties of 136
 - relationship to main diagram 135
 - restrictions on links 117
 - Subdiagram Properties dialog for 136
 - successful execution of 137
 - symbol 111, 136
 - unsuccessful execution 138
 - valid contents of 136
 - subdiagram or iterator
 - successfully ending 171
 - Subdiagram Properties dialog 136
 - subpackage 86
 - Substring by Position 328
 - Substring by Value 329
 - subtransaction step 131, 143
 - subtype
 - AppRequestNotYetSent exception 181
 - AppUnknown exception 179
 - ServiceCallTransportException exception 179
 - subtype of exception 424
 - successful execution status 143
 - Successfully ending a subdiagram or iterator 171
 - Successfully ending the main diagram 170
 - Sum 271
 - sum() method 396
 - summary
 - filter method 435
 - SmartCollabService method 445
 - StateManagement method 451
 - summary globals class method 441
 - Summary of collaboration template function blocks 273
 - Summary of database connection function blocks 285
 - summary of database stored procedure function blocks 299
 - Summary of date function block 311
 - Summary of string function blocks 325
 - Summary of the logging and tracing function blocks 317
 - summary of the utilities function blocks 333
 - summary, CwBidiEngine method 419
 - supported function blocks 157
 - Swap 272
 - swap() method 396
 - symbol
 - adding an action symbol to activity diagrams 113
 - aligning 183
 - center of 184
 - context menu for 110
 - deleting 146
 - description for 112
 - deselecting 109
 - display information for 22
 - edges 183
 - editing the properties of 112
 - End Failure 137, 144
 - End Failure symbol 111
 - End Success 137, 138, 143
 - End Success symbol 111
 - font for 21, 110
 - symbol (*continued*)
 - introduction to 110
 - iterator 139
 - label for 22, 112
 - moving 185
 - node symbols 111
 - nudging 185
 - orthogonal transition link 117
 - panning 186
 - properties of 21, 112, 191
 - selecting 109
 - selection border of 109
 - service call symbol 112
 - snapping to grid lines 22, 187
 - Start symbol 111
 - subdiagram 136
 - Subdiagram 111
 - text 142
 - transition link 117
 - transition link symbols 111
 - types of 110
 - zooming 22, 186
 - symbol properties dialog 191
 - Symbol Properties dialog 110, 112
 - symbols and links, changing the color 189
 - Symbols toolbar 23, 110
 - displaying 21
 - End Failure 144
 - End Success 143
 - select 142
 - text 142
 - synchronization 29
 - in business objects referenced by triggering business objects 40
 - of referenced business objects 40
 - using wrapper collaborations for 40, 48
 - synchronous service call 127
 - setting correlation attributes for 133
 - system configuration parameter
 - LOG_FILE 354, 364
 - MAX_LOG_FILE_SIZE 354, 364
 - MIRROR_LOG_TO_STDOUT 354, 364
 - NUMBER_OF_ARCHIVE_LOGS 354, 364
 - System Manager 11
 - System Manager showing default values for Database Resilience 74
 - system-generated trace message 195, 197
 - system-generated variable 91
 - currentException 167, 173
 - triggeringbusobj 205, 206
 - triggeringBusObj 35, 36
 - SystemException 284
 - SystemException exception 168, 358, 426
- ## T
- tags for activity definitions 156
 - template
 - collaboration 198
 - CollaborationFoundation 29
 - CustomerPartnerWrapper 50
 - ItemWrapper 51
 - standard collaboration 29
 - WrapperFoundation 47
 - template definition
 - creating 82

- template definitions window
 - declarations tab 198, 200
 - general tab 199
- Template Definitions window 17
 - Declarations tab 86
 - general description of 83
 - General tab 83
 - Ports and Triggering Events tab 94
 - Properties tab 91
- template function block
 - collaboration 273
 - summary of collaboration 273
- Template menu 22
- template tree view 16, 21
- template variable
 - affected by port name change 96
 - data types for 90
 - declaring 86, 90, 198
 - defined 86, 480
 - editing 86
 - port 96
 - port variables 91
 - special considerations for long-lived business processes 90
 - system-generated 91
 - used with correlation attributes 132
- termination 143
- Test Connector tool 12, 107
- Text Equal 329
- Text Equal Ignore Case 329
- Text Length 330
- The BiDiTransformation parameter in the Connector Configurator 71
- threads 55
- timeout value
 - dynamic 94
 - specifying for service calls 130
- To Array 339
- To String 258, 272, 306
- tool
 - Flow Manager 170
- toolbar
 - symbols 110
- toolbars
 - See Process Designer toolbars
- Tools menu 153
- toString() method 167, 382, 384, 397, 426, 427
- Trace 321
- Trace ID 1 321
- Trace ID 2 322
- Trace ID 3 322
- trace level 195, 197, 352
- trace message 195, 197
 - adding 195
 - assigning trace level to 195
 - collaboration-generated 195
 - generating 196, 363
 - setting trace level for 195
 - system-generated 195, 197
 - types of 195
- Trace on Level 323
- trace() method 195, 231, 363
- tracing 195, 197
 - code example 196
 - collaboration-generated 195
 - configuration of 195
 - generating message 196
 - level for 195
- tracing (continued)
 - system-generated 197
- Traditional Chinese 339
- transaction
 - recovery of 170, 180
- transaction levels 84
- transactional collaboration 128, 131, 143
 - defined 480
 - transaction levels 84
- TransactionException 284
- TransactionException exception 168, 358, 426
- transactions
 - beginning 225, 227, 399
 - committing 225, 227, 228, 400
 - defined 224
 - determining if in progress 228, 406
 - explicit 224
 - implicit 224
 - inheriting 226
 - managing 217, 224
 - rolling back 225, 227, 409
 - scope 225
- transactios
 - rolling back 170
- transition link 116
 - cancelling 117
 - connecting 120
 - creating 117
 - defined 111, 480
 - defining properties of 118
 - description 118
 - determining if valid 117
 - disconnecting 120
 - functions of 116
 - guidelines for using orthogonal and free-form links 117
 - label for 118
 - labeling 118
 - Link Properties dialog 118
 - modifying 120
 - number per node type 117
 - specifying a business object probe for 119
 - symbol 117
 - types of properties 118
- transport-related exceptions
 - handling boot-time recovery 180
 - handling runtime 179
- triggering access call 9, 94, 96, 97
- triggering business object 30
 - filtering data in 32
- triggering event 9, 94
 - assigning to scenario 97
 - copying 205
 - defined 94, 480
 - for called collaboration 52
 - initiating a map 201, 202
 - processing 35
 - simulating 107
 - variable for 205
 - when deleted 96
- triggeringbusobj system-generated variable 205, 206
- triggeringBusObj system-generated variable 35, 36, 91, 480
- Trim Left 330
- Trim Right 330
- Trim Text 330
- type
 - boolean data 208, 210, 224
 - date data 224

type (*continued*)
double data 208, 210, 224
float data 208, 210, 224
int data 208, 210, 224
long data 208, 210, 224
longtext data 208
string data 208, 210, 224

U

UID (symbol) 112, 480
End Failure 145
End Success 144
subdiagram 136, 137, 139
viewing 22
UML 5, 100
exporting UML files from a template 102
importing UML files to a template 102
unresolved flow 138, 170
defined 481
Unsent service call requests 181
Update request 92, 97, 131
update statement 217
UPDATE statement 287, 289, 293, 402, 403, 405
updateBO() 457
updateState() 457
updating previous versions of templates 103
Upper Case 330
Use_Retrieve process 462
USE_RETRIEVE property 473
user-defined variable 198
UserCollaborations package 86
using web services in activity definitions 116
utilities function block 333
Add Element 333
Catch Error 334
Catch Error Type 334
Condition 334
English 334
French 334
German 335
Get Country 335
Get Element 335
Get Language 335
Italian 336
Iterate Vector 336
Japanese 336
Korean 336
Loop 337
Move Attribute in Child 337
New Locale 337
New Locale with Language 338
New Vector 338
Raise Error 338
Raise Error Type 338
Simplified Chinese 338
Size 339
To Array 339
Traditional Chinese 339

V

Valid Data 258
validData() method 383
value
null attribute 207, 212

values for Database Resilience, System Manager showing
default 74
variable
CLASSPATH environment 166
currentException system-generated 167, 173
JCLASSES environment 165
populating the collaboration 203
triggeringbusobj system-generated 205, 206
triggeringBusObj system-generated 35, 36
user-defined 198
vector class
with executestoredprocedure() 221
with nextrow() 214
Vector class
with executeStoredProcedure() 301, 403, 412
with nextRow() 407
verb
in flow trigger 9
in triggering event 98
retrieving 375
setting 381
Verb
Create 259
Delete 259
Retrieve 260
Update 260
verification
of referenced business objects 40
using wrapper collaborations for 47
View menu 21, 152

W

warning message 193, 353
web services
including 53
web services function blocks 155
web services in activity definitions
using 116
Windows menu 23
workspace 183
grid 22, 186
wrapper collaboration 39
for maintaining data consistency 40
WrapperFoundation template 47
WrapperFoundation template 47
CustomerPartnerWrapper template 50
DestinationAppRetrieve port 49
features 47
From port 49
ItemWrapper template 51
modifying 50
ports 48
SourceApp port 49
To port 50
using 48
using for synchronization processing 48
using for verification processing 47

Y

yyyy-MM-dd 314
yyyyMMdd 315
yyyyMMdd HH
mm
ss 315

Z

- zero-length string 376
- Zoom/Pan toolbar 21, 23



Printed in USA